(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

CORRECTED VERSION

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
19 May 2005 (19.05.2005)

PCT

(10) International Publication Number
WO 2005/045692 A3

(51) **International Patent Classification**[7]**:** G06F 15/80, 15/78

(21) **International Application Number:** PCT/EP2004/009640

(22) **International Filing Date:** 30 August 2004 (30.08.2004)

(25) **Filing Language:** English

(26) **Publication Language:** English

(30) **Priority Data:**

03019428.6	28 August 2003	(28.08.2003)	EP
0302591 1.3	5 November 2003	(05.1 1.2003)	EP
103 57 284.8	5 December 2003	(05.12.2003)	DE
03028953.2	17 December 2003	(17.12.2003)	EP
03079015.8	17 December 2003	(17.12.2003)	EP
04002604.9	5 February 2004	(05.02.2004)	EP
04002719.5	6 February 2004	(06.02.2004)	EP
04003258.3	13 February 2004	(13.02.2004)	EP
04004885.2	2 March 2004	(02.03.2004)	EP
04075654.6	2 March 2004	(02.03.2004)	EP
04005403.3	8 March 2004	(08.03.2004)	EP
04013557.6	9 June 2004	(09.06.2004)	EP
04018267.7	2 August 2004	(02.08.2004)	EP
04077206.3	2 August 2004	(02.08.2004)	EP

(71) **Applicant** *(for all designated States except US):* **PACT XPP TECHNOLOGIES AG** [DE/DE]; Muthmannstrasse 1, 80939 Munchen (DE).

(72) **Inventors; and**
(75) **Inventors/Applicants** *(for US only):* **VORBACH, Martin** [DE/DE]; Gotthardstrasse 117A, 80689 Munchen (DE). **THOMAS, Alexander** [DE/DE]; c/o PACT XPP Technologies AG, Muthmannstrasse 1, 80939 Munchen (DE).

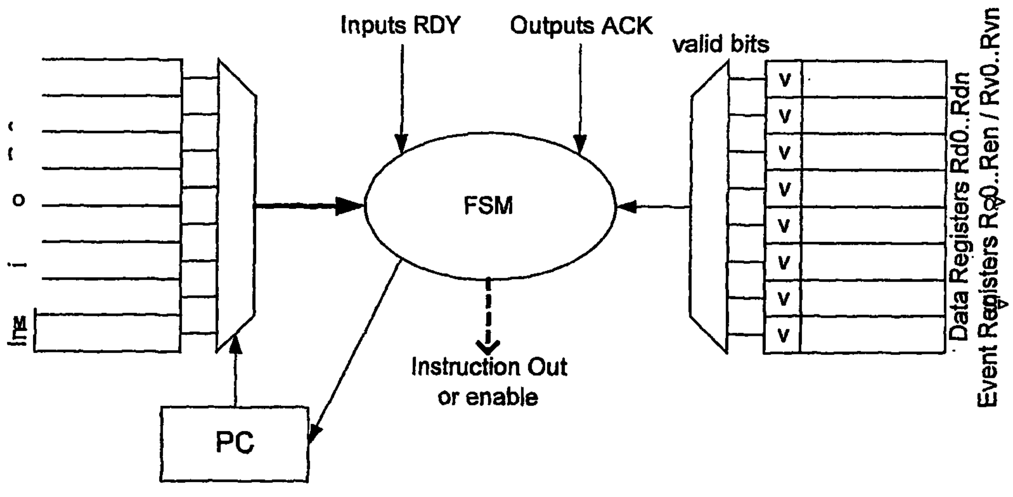
(74) **Agent: PIETRUK, Claus, Peter;** Heinrich-Lilienfein-Weg 5, 76229 Karlsruhe (DE).

(81) **Designated States** *(unless otherwise indicated, for every kind of national protection available):* AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW

(84) **Designated States** *(unless otherwise indicated, for every kind of regional protection available):* ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI,

[Continued on next page]

(54) **Title:** DATA PROCESSING DEVICE AND METHOD


Inputs RDY
Outputs ACK
valid bits
FSM
V
Instruction Out or enable
PC
Data Registers Rd0..Rdn
Event Registers Re0..Ren / Rv0..Rvn

(57) **Abstract:** A reconfigurable data processing device comprising a multidimensional array of coarse grained logic elements processing data and operating at a first clock rate and communicating with one another via communication lines operated at a second clock rate, wherein the first clock rate is higher than the second and wherein the coarse grained logic elements comprise storage means for storing data needed to be processed.

WO 2005/045692 A3

FR, GB, GR, HU, IE, IT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*

(88) Date of publication of the international search report: 2 March 2006

(48) Date of publication of this corrected version: 23 March 2006

(15) Information about Correction:
see PCT Gazette No. 12/2006 of 23 March 2006, Section II

Data processing device and method

The present invention relates to reconfigurable computing. In particular, the present invention relates to improvements in the architecture of reconfigurable devices.

Reconfigurable data procesing arrays are known in the art. Reference is being made to the previous applications and/or publications of the present applicant /assignee all of which are encorporated herein by way of reference. Accordingly, the devices described hereinafter may be multidimensional (n>1) arrays comprinsing coarse grained computing and/or data operation elements allowing for runtime reconfiguration of the entire array or parts thereof, preferably in response to a signal indicating reconfigurability to a loading unit (CT, CM or the like) .

Now, several of these data procesing arrays have been built (i.e. Xppl, XPP128, XPP2, XPP64) . It is however desirable to improve the known device further as well as to improve methods of„its operation.

Accordingly, in order to achieve this object there will be described a number of improvements allowing separately or in commoxn to improve the performance and /or power consumption and /or cost of the device.

A first way to improve the known devices is to improve the functionability of each single processor element. It has been previously suggested to include a ring-memory (RINGSPEICHER) in the array, to store instructions in the ring-memory and to provide a pointer that points to one of the ring-memory adreses so as to select an instruction to be carried out next. Furthermore, it has been suggested to provide at least one „shadow configuration" and to switch over between several configurations /shadow configurations. Anotrher or additional suggestions has been designated as ,,wave reconfiguration".

While these known methods improve the performance of a reconfigurable device, there seems to be both a need and a possibility for further improvements.

It is to be understood that while in the following description, a detailed example is given, for example with respect to the number of registers given associated with eachj PAE, it is not deemed necessary to provide an ALU with exactly this number of registers. Rather, it will be understood by the average skilled person that deviations from the explicitly described embodiment are easily feasible and that the detailed level of description stems from an effort to provide an exemplary PAE and not from the wish to restrict the scope of invention. -

1 Overyiew of changes vs. XPP XPP-II

1.1 ALXJ-PAE Architecture

In the suggested improved architecture, a PAE might e.g. comprise 4 input ports and 4 output ports. Embedded with each PAE is the FREG path newly named DF with its dataflow capabilities, like *MERGE, SWAP, DEMUX* as well as *ELUT.*

2 input ports RiO and RiI are directly connected to the ALU. Two output ports receive the ALU results.
Ri2 and Ri3 are typically fed to the DF path which output is Ro2 and Ro3 .
Alternatively Ri2 and Riβ can serve as inputs for the ALU as well. This extension is needed to provide a suitable amount of ALU inputs if Function *Folding* (as described later) is used. In this mode Ro2 and. Ro3 serve as additional outputs.

Associated to each data register (Ri or Ro) is an event port (Ei or Eo) .

It is possible, albeit not necessary to implement an additional data and event bypass BRiO-I, BEiO-. The decision depends on how often Function Folding will be used and how many inputs and outputs are required in average.

(see Fig. 1 now)

1.1.1 Other extensions

SIMD operation is implemented in the ALUs to support 8 and 16 bit wide data words for i.e. graphics and imaging.

Saturation is supported for ADD/SUB/MUL instructions for i.e. voice, video and imaging algorithms.

1.2 Function Folding

1.2.1 Basics and input/output paradigms

Within this chapter the basic operation paradigms of the XPP architecture are repeated for a better understanding based on Petri-Nets. In addition the Petri -Nets will be enhanced for a better understanding of the subsequently described changes of the current XPP architecture.

In most arrays each PAE operates as a data flow node as defined by Perti-Nets.- (Some arrays might have parts that have other functions and should thus be not considered as a standard PAE).. A Petri-Net supports a calculation of multiple inputs and produces one single output. Special for a Perti-Net is, that the operation is delayed until all input data is available.

For the XPP technology this means :

1. all necessary data is available
2. all necessary events are available

The quantity of data and events is defined by the data and control flow, the availability is displayed at runtime by the handshake protocol RDY/ACK.

(see Fig. 52 now)

Here, the thick arbor indicates the operation, the dot on the right side indicates that the operation is delayed until all inputs are available.

Enhancing the basic methodology function folding supports multiple operations - maybe even sequential - instead of one, defined as a *Cycle*. It is important that the basics of Petri - Nets remain unchanged.

(see Fig. 53 now)

Here, typical PAE-like Petri-Nets consume one input packet per one operation. For sequential operation multiple reads of the same input packet are supported. However, the interface model again keeps unchanged.

Data duplication occurs in the output path of the Petri-Net, which does not influence the operation basics again.

(see Fig. 54 now)

1.2.2 Method of Function Folding

One of the most important extensions is the capability to fold multiple PAE functions onto on PAE and execute them in a sequential manner. It is important to understand that the intention is not to support sequential processing or even microcontroller capabilities at all. The intention of Function Folding is just to take multiple dataflow operations and map them on a single PAE, using a register structure instead of a network between each function.

One goal may be to save silicon area by rising to clock frequency locally in the PAEs. An additional expectation is to save power since the busses operate at a fraction of the clock frequencies of the PAEs. Data transfers over the busses, which consume much power, are reduced.

(see Fig. 2 now)

The internal registers can be implemented in different ways, e.g. -in one of the following two:

1 . dataflow model

Each register (r') has a valid bit which is set as soon as data has been written into the register and reset after the data has been read. Data cannot be written if valid is set, data can not be read if valid is not set. This approach implements a 100% compatible dataflow behaviour.

2 . sequencer model

The registers have no associated valid bits. The PAE operates as a sequencer, whereas at the edges of the PAE (the bus connects) the paradigm is changed to the XPP-like dataflow behaviour.

Even if at first the dataflow model seems preferable, it has major down sides. One is that a high amount of register is needed to implement each data path and data duplication is quite complicated and not efficient. Another is that sometimes a limited sequential operation simplifies programming and hardware effort .

Therefore it is assumed consecutively that sequencer model is implemented. Since pure dataflow can be folded using automatic tools the programmer should stay within the dataflow paradigm and not be confused with the additional capabilities. Automatic tools must take care i.e. while register allocation that the paradigm is not violated.

Fig. 3 now shows that using sequencer model only 2 registers (instead of 4) are required.

For allowing complex function like i.e. address generation as well as algorithms like "IMEC"-like data stream operations the PAE has not only 4 instruction registers implemented but 8, whereas the maximum bus-clock vs. PAE-clock ration is limited to a factor of 4 for usual function folding.

It is expected that the size of the new PAE supporting Function Folding will increase by max. 25%. On the other hand 4 PAEs are reduced to 1 .

Assuming that in average not the optimum but only about 3 functions can be folded onto a single PAE a XPP64 could be replaced by a XPP21. Taking the larger PAEs into account the functionality of a XPP64 XPP-II should be executable on a XPP XPP-III with an area of less than half.

The function folding method and apparatus as well as otrher further improvements will be described in even more detailed hereinafter.

Equality of internal data registers and bus transfers

The function fold concept realises two different models of data processing:

a) Sequential model, wherein within the PAE the same rules apply as in von-Neuman- and Harvard-processors.
b) PACT VPU-model, wherein data are calculated or operated upon in arbitrary order according to the PETRI -Net -Model (data flow + synchronisation) .

Due to the unpredictability of the arrival of data at the input registers (IR) a deadlock or at a least significant reduction in performance could occur if the commands in RCO... RCn were to be performed in a linear manner-. In particular, if feed-backs of the PAE outputs to the inputs of the PAE are present, deadlocks might occur. This can be avoided if the instructions are not to be processed in a given order but rather according to the possibility of their processing, that is, one instruction can be carried out as soon as all conditions of the VPU-model are fulfilled. Therefore, for example, once all RDY -handshake s of incoming data, ACK- handshakes of outgoing data and, if necessary, triggers (including their handshakes) are valid, then the instruction can be carried out. As the FF PAE has data additionally stored in internal registers, their validity and status has to be checkable as well in a preferred embodiment. Therefor, every internal data register (RDO... RDn) is separately assigned a valid bit indicating whether or not valid data are present in the register. When writing data into the register, valid is set, when reading, valid is reset. Data can be read only if "valid" is set and can be written only if "valid" is not set. Accordingly, the valid flag corresponds most closely to the status that is produced in the state machines of bus systems by the transmittal of RDY/ACK-handshakes. It is a preferred embodiment and considered to be inventive to provide a register with a status bit in that way.

It is therefore possible to carry out instructions at the time when all conditions for the execution - again very similar to PETRI -nets are fulfilled.

Basically, there are two methods available for selection of instruction and control of their execution described herein after.

Method A : FF PAE Program Pointer
(Finite State Machine & Program Pointer-Approach)

(see Fig. 4 now)

According to the control principle of sequential processors, a program counter is used to select a certain instruction within the instruction memory. A finite state machine controls the program counter.- This finite state machine now checks whether or not all conditions for the instruction in RC (PC), that is the instruction, onto which the PC (Program Counter) points, are fulfilled. To do so, the respective RDY- and/or ACK-handshakes of the in- and/or outputs needed for the execution of the instructions are checked. Furthermore, the valid- flags of the internal registers to be read (RDO..RDn) are checked so as to control whether or not they are set, and the valid- flags of those internal registers (RDO..RDn) into which is to be written, are checked whether they are not set. If one of the conditions is not fulfilled, the instructions will not be carried out. PC is controlled to count further, the instruction is skipped and the next instruction is selected and checked as described.

The advantage of this method is the compatibility with sequential processor models. The disadvantage resides in the necessity to test and to skip instructions. Both of which might result in significant losses of performance under certain circumstances.

Method B : FF PAE Program Pointer
(Enabler & Arbiter-Approach)

(see Fig. 5 now)

This method is based upon the possibility to test all instructions in Rc0..Ren in parallel. In order to save the expense of the complete decoding of array instructions, each RC is assigned an entry in an evaluation mask field, the length of which corresponds to the maximum number of states to be tested; therefore, for every possible RDY- or ACK-trigger-signal (as well the RDY/ACKs of the triggers) as well as for every valid bit in RD0...RDn two bits are available indicating whether or not the respective signal is to be set or not set; or, whether the state of the signal is unimportant for the execution of the instruction.

Example mask

InData-RDY		OutData-ACK		InTrigger			OutTrig-ger-ACK		Rd Data Valid	
Rdy value	don't care	Ack value	don't care	trigger value	rdy value	don't care	ack value	don't care	valid value	don't care

The mask shows only some entries. At In-Trigger, both the state of the trigger (set, not set) as well as the value of the trigger (trigger value) can be tested via RDY-value.

A test logic testing via for example the Line Control de¬scribed herein after all instructions in parallel. Using an arbiter, an instruction of the set of all executables is se¬lected. The arbiter controls the instruction multiplexer via ISeI according to the transferral of the selected instructions to the PAE.

The Line Control has one single line of Boolean test logic for every single instruction. By means of an ExOR-gate (e) the value of the signal to be tested against the setting in em of the line is checked. By means of an OR-gate (+) respectively, a selection is carried out, whether the checked signal is relevant (don't care) . The results of all checked signals are ANDed. A logic 1 at the output of the AND-gates (&) shows an executable instruction. For every RC, a different test-line exists. All test-lines are evaluated in parallel. An arbiter having one of a number of possible implementations such as a priority arbiter, Round-Robin-Arbiter and so forth, selects one instruction for execution out of all executable instruc-tions. There are further implementations possible obvious to the average skilled person. Those variants might be widely equivalent in the way of operation and function. In particu¬lar, the possibility of using "negative logic" is to be men¬tioned.

(see Fig. 6 now))

Fig. 7 now gives an overview of the entire circuitry.

Advantages of the method are :

- Significantly fast, in view of the fact that one instruc¬tion can be carried out in every single clock

- Reduced power consumption, since no energy is wasted on disgarded cycles which is in particular advantageous to the static power dissipation.
- Similar hardware expense as in the sequential solution when using small and medium sized configuration memories (RC) therefor similar costs.

Disadvantages :

- Likely to be significantly more expensive on large RC; therefore, an optimisation is suggested for a given set of applications .
- In order to implement the sequencer mode (compare other parts of the application) the program counter having an FSM must be provided for. The FSM then is restricted to the tasks of the sequencer so that the additional expenses and the additional costs are relatively low.

Depopulated Busses according to the State of the Art

All busses assigned to a certain PAE are connected to the input registers (IR) or the output registers of the PAE are connected to all busses respectively (compare for example DE 100 50 442.6 or the XPP/VPU-handbooks of the applicant).

It has been realised that PAEs, in particular FF PAEs, allow for a depopulation of bus interconnects, in particular, if more IR/OR will be available compared to the State of the Art of the XPP as previously known. The depopulation, that is the reductions of the possibilities to connect the IR or ER onto the busses can be symmetrically or asymmetrically. The depopulation will typically amount to 20 to 70 %. It is significant that the depopulation will not or not significantly effect the interconnectability and/or the routability of an algorithm in a negative way.

The method of depopulation is particularly relevant in view of the fact that several results can be achieved. The hardware-expense and thus the costs of the bus systems can be reduced significantly; the speed of the busses is increased since the gate delay is reduced by the minimisation of connecting points; simultaneously, the power consumption of the busses is reduced.

A preferred depopulation according to the VPU-architecture according to the State of the Art, however, with more IR/OR is shown in Fig. 8 now.

In particular, reference is being made to an optional extension of the bus architecture allowing for a direct next neighbour data transfer of two adjacent PAEs, in particular two PAEs placed one onto the other. Here, the outputs (OR) of one PAE are directly connected to a dedicated bus which is then directly connected to the inputs (IR) of a neighbouring PAE (compare Fig. 9 now). The figure only shows an horizontal next neighbour bus, however, in general, vertical busses are possible as well.

In Fig. 8 now, the shaded circles stand for possible bus connects: MUX. Double circuits stand for a connection from the bus: DeMUX.

Changes of the PAE ID

Fig. 9 now shows the State of the Art of a PAE implementation as known from XPU128, XPP64A and described in DE 100 50 442.6

The known PAE has a main data flow in the direction from top to bottom to the main ALU in the PAE-core. At the left and right side, data channels are placed additionally transmitting data along the main data flow direction, once the same direction as the main data flow (FREG) and once in the reverse direction (BREG). On both sides of the PAE, data busses are provided that run in the reverse direction of the main data flow of the PAE and onto which the PAE as well as FREG and BREG are connected. The architecture of the State of the Art requires eight data busses for each PAE side as well as four transfer channels for FREG/BREG for typical applications.

The bus system of the State of the Art has switching elements, register elements (R), each at the side of the PAEs. The switching elements allow for the disruption of a bus segment or disconnection to a neighbouring bus, the register elements allow the construction of an efficient pipelining by transferring data through the register, so as to allow for higher transferral band-width. The typical latency in vertical direction for next -neighbour- transmitting is 0 per segment, however is 0,5-1 in horizontal direction per segment and higher frequencies.

(see Fig. 10 now)

Now, a modified PAE structure is suggested, wherein two ALUs, each having a different main data flow direction are provided in each PAE, allowing for significantly improved routability. On one hand, the tools used for routing are better and simpler, on the other hand, a significant reduction in hardware resources is achieved. First tests shows that the number of busses necessary in horizontal direction is reduced by about 25 % over the State of the Art. The vertical connects in FREG/BREG (= BYPASS) can even be reduced by about 50 %. Also, it is no more necessary to distinguish between FREG and BREG as was necessary in DE 100 50 442.6.

(see Fig. 11 now)

The double -ALU ;structure has been further developed to an ALU-PAE having Inputs and outputs in both directions. Using automatic routers as well as hand-routed applications, further additional significant improvements of the network topology can be shown. The number of busses necessary seems to be reduced to about 50 % over the State of the Art, the number of vertical connects in the FREG/BREG (= BYPASS) can be reduced by about 75 %.

(see Fig. 12 now)

For this preferred embodiment which can be used for conventional as well as for function fold ALUs, it is possible to place register and switching elements in the busses in the middle of the PAE instead of at the sides thereof (see Fig. 13 now).

In this way, it is possible even for high frequencies to transmit data in horizontal direction to the respective neighbouring PAE without having to go through a register element. Accordingly,; it is possible to set up next neighbour connections in vertical and horizontal directions which are latency free (compare State of the Art and drawings referring to depopulated busses). The example of the interconnections shown in the respective figure allows transferral having zero latency in vertical direction and horizontally from left to right. Using an optimisation of PAE interface structure a latency free next neighbouring transmission in both horizontal directions can be achieved. If in every corner of the PAE input register (IR, arrow of bus into PAE) from bus and output register (OR, arrow from PAE to bus) to the bus are implemented, each neighbouring PAE can exchange data without latency.

(see Fig. 14 now)

It is possible to further optimise the above disclosed PAE arrangement. This can be done by using no separate bypass at all in all or some of the PAEs. The perferred embodiment comprises two ALUs, one of these being "complete" and having all necessary functions, for example multiplication and BarrelShift while the second has a reduced instruction set eliminating functions that require larger arrays such as multiplication and BarrelShift. The second ALU is in a way replacing BYPASS (as drawn). There are several possible positions for the register in switching elements per bus system, and two of the preferred positions per bus are shown in Pig. 15 in dotted lines.

Both ALUs comprise additional circuits to transfer data between the busses so as to implement the function of the bypass. A number of possible ways of implementations exist and two of these shall be explained as an example.

a) Multiplexer

Configurable multiplexers within the ALU are connected so that ALU inputs are bypassing the ALU and are directly connected to their outputs.

b) MOVE instruction

A MOVE instruction, stored in RcO..Ren is transferring within the respective processing clock of the function fold the data according to the input specified within the instruction to the specified output.

Superscalarity/ Pipelining

It is possible and suggested as first way of improving performance to provide roughly superscalar FF ALU-PAEs which calculate for example 2,4,8 operations per bus clock @ FF=2,4,8, even while using the MUL opcode.

The basic concept is to make use of the VALID- flags of each internal register. MUL is implemented as one single opcode which is pipelined over two stages.

MUL takes its operands from the input registers Ri and stores the results into internal data registers Rd. VALID is set if data is stored into Rd. ADD (or any other Opcode, such as BSFT) uses the result in Rd if VALID is set; if not the execution is skipped according to the specified VALID behaviour.
In addition the timing changes for all Opcodes, if the MUL instruction is used inside a PAE configuration. In this case all usually single cycle OpCodes will change to pipelined 2 cycle OpCodes. The change is achieved by inserting a bypass able multiplexer into the data stream as well as into control.

The following program will be explained in detail:

```
MUL (RdO, RdI), RiO, RiI;
ADD RoO, RdI, Ri2 ;
```

In the first bus-cycle after configuration (t_0) MUL is executed (assuming the availability of data at RiO/1) . The register pair Rdθ/1 is invalid during the whole bus-cycle, which means during both FF-PAE internal clock cycles. Therefore ADD is not executed in the 2nd clock cycle. After t_0 the result of MUL is written into the register pair, which VALID flags are set at the same time.

In ti new data is multiplied. Since the VALID is set for Rdθ/1 now the ADD command is executed in the 2nd clock cycle, but takes 2 clock cycles for over all execution. Therefore operand read and result write is inline for both operations, MUL as well as ADD.

The result of a MUL-ADD combination is available with 2 clocks latency in a FF=2 ALU-PAE. For FF >= 6 no latency is inserted.

(see Fig. 16 now)

However since multiplication and all other commands are processed in parallel the machine streams afterwards without any additional delays.

(see Fig. 17 now)

If there are OpCodes besides MUL which require 2 clock cycles for execution (e.g. BSTF) the architecture must be modified to allow at least 3 data writes to registers after the second internal clock cycle.

The data path output multiplexer gets 2 times larger as well as the bus system to the output registers (OR) and the feedback path to the internal data registers (Rd) .
If accordingly defined for the OpCodes, more than 4 internal registers can be used without increasing the complexity by using enables (en) to select the specific register to write in the data. Multiple registers are connected to the same bus,

e.g. RdO, Rd4, Rd8, RdI2. However not all combinations of register transfers are possible with this structure. If e.g. MUL uses RdO and RdI the following registers are blocked for the OpCode executed in parallel : Rd4, 5, 8, 9, 12, 13.

Register map:

Rd0	Rd4	Rd8	Rd12
Rd1	Rd5	Rd9	Rd13
Rd2	Rd6	Rd10	Rd14
Rd3	Rd7	Rd11	Rd15

Datapath architecture: see Fig. 18 now.

The Sequencer PAEs

Since there is a need to be able to run control flow dominated applications on the XPP III as well, Sequencer PAEs will be introduced. Such a PAE can be thought of as a very simple kind of processor which is capable to run sequential code within the XPP. This allows the efficient implementation of control flow oriented applications like the H.264 Codec on the array whereas with SEQ-PAEs missing the realization would be more difficult and resource consuming.

The SEQ-PAEs are not build from scratch. Instead such a tile will be build up by a closely coupling of a ALU-PAE and neighboring RAM-PAE, which can be seen in Fig. 19 now.

Therefore the functionality of the ALU- as well as RAM-PAE has to be enhanced to be able to fulfill the requirements of such a SQE-PAE. This information will be given next.

ALU-PAE Enhancements

The extended version of the ALU-PAE is given in Fig. 20 now. To the right border the registers which are controlling the different modules can be seen. Those registers will be used in normal- as well as in SEQ-mode. Therefore the appropriate control signals from the local configuration manager and the RAM-PAE are first merged by OR-Gates and then are forwarded to the register whereas it has to be ensured that in normal mode the signals from the RAM-PAE are 0 and vice versa.

Further more, since the ALU-PAE marks the execution part of the tiny processor, there is a need to transfer values to and from the internal register directly to the RAM. Therefore a additional multiplexer AMI is inserted in the multiplexer hierarch of section 2. In the normal mode this multiplexer feeds the word from its predecessor to the next stage whereas in the SEQ mode an immediate value provided by the Imm. Register will be delivered. In addition in SEQ mode a value of one of the internal .registers can be delivered to the RAM-PAE via the output of the multiplexer.. However, it has also **to be considered to provide a "LOAD reg, imm" since this is not much slower than ,,ADD reg, reg, imm"**

To enable the RAM-PAE to write data to the internal register of the ALU-PAE another multiplexer is inserted in the multiplexer chain .of section 4 . Similar to the scenario given above this multiplexer will only be activated in SEQ mode whereas in normal mode this multiplexer will just forward the data of its predecessor. In one preferred embodiment, it is suggested to place **RS2 behind BSFT-Mux in view of the delay. Data could be written into the internal registers via this. (LOAD reg, imm)]**

As it has already been discussed, data can be processed during one or two cycles by the ALU-PAE depending on the selected arithmetic function. Due to the auto synchronization feature of the XPP and due to the fact that in normal mode a successive operation will not start before the previous one is finished, it does not really care if an operation lasts one or two clock cycles. Whereas the tile is working in SEQ mode there is a difference since we assume to have a pipeline character. This means that a one cycle operation could run in parallel with a two cycle module where the operation would be executed in stage two at this time. Due to the limited multiplexing capacities of a word - 16 Bit - only one- result could be written to the connected registers whereas the other one would be lost. In general there are three possibilities to solve this problem.

The first one could be that the compiler is capable to handle this problem. This would mean that it has to know about the pipeline structure of the whole SEQ-PAE as well as of a tile in detail. To prohibit a parallel execution the compile would have to add a NOP to every two cycle instruction for the structure given above. However this idea seems not to be convenient due to the strong relation between the hardware structure and the compiler. The drawback would be that every time changes are made to the hardware the compile would most likely have to be trimmed to the new structure.

The second idea could be to recognize such a situation in the decode stage of the pipeline. If a two cycle instruction is directly followed by an instruction accessing a one stage arithmetic unit it has to be delayed by one clock cycle as well .

The last possibility is to make the complete ALU-PAE look like a two stage execution unit . Therefore only one register has to be included in the multiplexer chain of section four right after the crossover from the multiplexer separating the one stage of the two stage modules. Obviously, this is preferred.

Comparing the last to ideas the third one seems to be the best one since only one register has to be inserted. If we a closer look to the second solution special logic would be needed for analyzing the disallowed combination of instructions as well as logic for stopping the program counter (PC) and the instruction retardation. It has to be assumed that this logic would require much more area than the registers as well as the fact that the delay of the logic would possibly increase the critical path.

Since it has to be distinguished between the SEQ and the normal mode where a one cycle execution should still be available. This possibility is given by a multiplexer which allows to bypass the RS2 Register as shown in the corresponding figure (Fig. 20 now).

The RAM-PAE

A short description of the stages

To get- the SEQ-PAE working there still has to be provided more functionality. Right now the RAM-PAE will take care of it. As a first approach for realizing the sequencer a four stage pipeline has been chosen. The stages are, as it can be seen in Fig. 21 now:

- The fetch stage
- The decode stage
- The execution stage 1
- The execution stage 2

In the fetch stage the program counter for the next clock cycle will be calculated. This means that it will be either incremented by 1 via a local adder or one of the program counters from the decode or execution stage 2 will be selected. The program counter of the execution stage thereby provides the address if a call instruction occurred whereas the program counter of the execution stage provides the PC if there has been a conditional jump. Right now the branch address can either be calculated out of the current PC and a value which either be an immediate value or a value from a internal registers of the ALU-RAM - indirect addressing mode - or an absolute value. This e.g. is necessary if there is return from a subroutine to the previous context whereas the according absolute PC will be provided by the stack bank.
In the decode stage the instruction coming from the code bank will be decoded. Necessary control signals and, if needed, the immediate value for the internal execution stage 1 as well as for the execution stage 1 of the ALU-PAE will be generated.

The signals include the control information for the multiplexers and gating stages of section two of the ALU-PAE, the operation selection of the ALU's tiles, e.g. signed or unsigned multiplication, and the information whether the stack pointer (SP) should be in/decremented or kept unchanged in the next stage depending on the fact if the instruction is either a call or jump. In case a call instruction occurred a new PC will be calculated in parallel and delivered to the fetch stage .

Furthermore the read address and read enable signal to the data bank will be generated in case of a load instruction.
In the execution stage 1, which by the way is the first stage available on the ALU as well as on the RAM-PAE, the control signals for execution stage 2 of the ALU-PAE are generated. Those signal will take care that the correct output of one of the arithmetical tiles will be selected and written to the enabled registers. If the instruction should be a conditional jump or return the stack pointer will be modified in this stage. In parallel the actual PC will be saved to the stack bank at the address give by the Rsp EX1 register in case of a branch. Otherwise, in case of a return, the read address as well as the read enable signal will be applied to the stack bank.

In execution stage 2 the value of the PC will be calculated and provided to the multiplexer in the fetch stage in case of a jump. At the time write address and write enable signal to the data bank are generated if data from the ALU have to be saved.
Instead of two adders, it is possible to provide only one in the rpp path. .

Pipeline actions

In the following section a short overview of the actions that are taking place in the four stages will be given for some basic instructions. It should help to understand the behaviour of the pipeline. Since the instruction which is going to be discussed will be available at the instruction register the actins of the fetch stage will be omitted in this representation.

IR: Instruction Register
DR: Data Register
DB: Data Bank
SBR : Store/Branch Register

Instruction: Load value from data bank to R[n]

ALU-PAE	RAM-PAE
decode stage	

	IR_ex1 <- IR_ex2
	Control Registerset EXS1 <- 0x0
	Imm. EXS1 <- 0x0
	Rpp_ex1 <- Rpp_de
	DB_radr <- imm

Execution stage 1	

	IR_ex2 <- IR_ex1
	Control Registerset EXS2 <- enable R, set mux section 4
	Rpp_ex2 <- Rpp_ex1
	DR <- DB_radr [imm]
	Rsp_ex2 <- Rsp_ex1

Executinon stage 2	
R[n] <- DR	

Instruction: Sore value from R [n] to data bank

ALU-PAE	RAM-PAE
decode stage	

	IR_ex1 <- IR_ex2
	Control Registerset EXS1 <- enable mux section 2
	Imm. EXS1 <- 0x0
	Rpp_ex1 <- Rpp_de

Execution stage 1	

SBR <- R[n]	IR_ex2 <- IR_ex1
	Control Registerset EXS2 <- 0x0
	Rpp_ex2 <- Rpp_ex1
	Rsp_ex2 <- Rsp_ex1

Executinon stage 2	

	DB_wradr <- imm
	DB_wrdata <- SBR

1.3 Array Structure

First advantages over the prior art are obtained by using function folding PAEs. These as well as other PAEs can be improved .

The XPP-II structure of the PAEs consumes much area for FREG and BREG and their associated bus interfaces. In addition feed backs through the FREGs require the insertion of registers into the feedback path, which result not only in an increased latency but also in a negative impact onto the throughput and performance of the XPP.

A new PAE structure and arrangement is proposed with the expectation to minimize latency and optimize the bus interconnect structure to achieve an optimized area.

The XPP-III PAE structure does not include BREGs any more. As a replacement the ALUs are alternating flipped horizontally which leads to improved placement and routing capabilities especially for feedback paths i.e. of loops.

Each PAE contains now two ALUs and two BP paths, one from top to bottom and one flipped from bottom to top.

(see Fig. 22 now)

1.4 Bus modifications

Within this chapter optimizations are described which might reduce the required area and the amount of busses. However, those modifications comprise several proposals, since they have to be evaluated based on real algorithms. It is possible to e.g. compose a questionnaire to collect the necessary input from the application programmes .

1.4.1 Next neighbour

In XPP-II architecture a direct horizontal data path between two PAEs block a vertical data bus. This effect increases the required vertical busses within a XPP and drives cost unnecessarily. Therefore in XPP-III a direct feed path between horizontal PAEs is proposed.

In addition horizontal busses of different length are proposed, i.e. next neighbour, crossing 2 PAEs, crossing 4 PAEs.

1.4.2 Removal of registers in busses

In XPP-II registers are implemented in the vertical busses which can be switched on by configuration for longer paths. This registers can furthermore be preloaded by configuration which requires a significant amount of silicon area. It is proposed to not implement registers in the busses any more, but to use an enhanced DP or Bypass (PB) part within the PAEs which is able to reroute a path to the same bus using the DP or BP internal registers instead.

(see Fig. 23 now)

Here, it might be to decide how many resources are saved for the busses and how many are needed for the PAEs and /or how *often must registers be inserted, are 1 or max. 2 paths enough per PAE (limit is two since DF/BP offers max. 2 inputs*

1.4 .3 "Shifting n:1, 1:n capabilities from busses to PAEs

In XPP-II n:1 and 1:n transitions are supported by the busses which requires a significant amount of resources i.e. for the sample -and-hold stage of the handshake signals.

Depending on the size of n two different capabilities are provided with the new PAE structure:

$n \leq 2$	The required operations are done within the DF path of the PAE
$2 \leq n \leq 4$	The ALU path is required since 4 ports are necessary,
$n > 4$	Multiple ALUs have to be combined.

This method saves a significant amount of static resources in silicon but requires dedicated PAE resources at runtime.

Here, it might be worthwhile to evaluate how much silicon area is saved per bus how often occurs $n=2$, $2 \leq n \leq 4$, $n > 4$ the ratio between saved silicon area and required PAE resource and to decide on the exact bus structure in repsonse to one or all of said criteria.

1.5 FSM in RAM-PAEs

In the XPP-II architecture implementing control structures is very costly, a lot of resources are required and programming is quite difficult.

However memories can be used for a simple FSMs implementation. The following enhancement of the RAM-PAEs offers a cheap and easy to program solution for many of the known control issues, including HDTV.

(see Fig. 24 now)

Basically the RAM-PAE is enhanced by an feedback from the data output to the address input through a register (FF) to supply subsequent address within each stage. Furthermore additional address inputs from the PAE array can cause conditional jumps, data output will generate event signals for the PAE array. Associated counters which can be reloaded and stepped by the memory output generate address input for conditional jumps (i.e. end of line, end of frame of a video picture) .
A typical RAM-PAE implementation has about 16-32 data bits but only 8-12 address bits. To optimize the range of input vectors

it is therefore suggested to insert some multiplexers at the address inputs to select between multiple vectors, whereas the multiplexers are controlled by some of the output data bits.

One implementation for an XPP having 24bit wide data busses is sketched in Fig. 25 now. 4 event inputs are used as input, as well as the lower for bits of input port RiO. 3 counters are implemented, 4 events are generated as well as the lower 10 bits of the RoO port.

The memory organisation suggested here may be as follows:

- 8 address bits
- 24 data bits (22 used)
  - 4 next address
  - 8 multiplexer selectors
  - *6* counter control (shared with 4 additional next address)
  - 4 output

(see Fig. 25 now)

It is to be noted that the typical memory mode of the RAM-PAE is not sketched in the block-diagram mentioned above.,

The width of the counters is according to the bus width of the data busses.

For a 16 bit implementation it is suggested to use the carry signal of the counters as their own reload signal (auto re-load) , also some of the multiplexers are not driven by the memory but "hard wired" by the configuration.

The proposed memory organisation is as follows:

- 8 address bits
- 16 data bits (16 used)
  - 4 next address
  - 4 multiplexer selectors
  - 3 counter control (shared with 3 additional next address)
  - 4 output

(see Fig. 26 now)

It *is to be noted that actually the RAM-PAEs typically will not be scaleable any more since the 16-bit implementation is* *different from the 24-bit implementation . It is to decide whether the striped down 16-bit implementation is used for 2.4-bit alB .*

1.6 IOAG interface

1.6.1 Address Generators and bit reversal addressing

Implemented within the IO interfaces are address generators to support e.g. 1 to 3 dimensional addressing directly without any ALU-PAE resources. The address generation is then done by 3 counters, each of them has e.g. configurable base address, length and step width.
The first counter (CNT1) has a step input to be controlled by the array of ALU-PAEs. Its carry is connected to the step input of CNT2, which carry again is connected to the step input of CNT3 .
Each counter generates carry if the value is equal to the configured length. Immediately with carry the counter is reset to its configured base address.

One input is dedicated for addresses from the array of ALU-PAEs which can be added to the values of the counters. If one or more counters are not used they are configured to be zero.

In addition CNT1 supports generation of bit reversal addressing by supplying multiple carry modes.

(see Fig. 27 now)

1.6.2 Support **for different** word width

In general it is necessary to support multiple word width within the PAE array. 8 and 16 bit wide data words are preferred for a lot of algorithms i.e. graphics. In addition to the already described SIMD operation, the IOAG allows the split and merge of such smaller data words.

Since the new PAE structure allows 4 input and 4 output ports, the IOAG can support word splitting and merging as follows:

I/O 0	I/O 1	I/O 2	I3
16/24/32-bit data word			address
16-bit data word	16-bit data word		address
8-bit data word	8-bit data word	8-bit data word	address

Input-ports are merged within the IOAG for word writes to the IO.
For output-ports the read word is split according to the configured word width.-

1.7 Multi-Voltage Power Supply and Frequency Stepping

PAEs and busses are build to perform depending on the workload. Therefore the clock frequency is configurable according to the data bandwidth, in addition clock gating for registers is supported, busses are decoupled using row of AND gates. Dynamically clock pulses are gated, whenever no data can be processed.

Depending on the clock frequency in the PAEs and the required bandwidth for the busses the voltage is scaled in an advanced architecture. Within the 4S project such methods are evaluated and commercially usable technologies are researched.

1.8 XPP / µP coupling

For a closed coupling of a µP and a XPP a cache and register interface would be the preferable structure for high level tools like C-compilers. However such a close coupling is expected not to be doable in a very first step.

Yet, two different kind of couplings may be possible for a tight coupling:

a) memory coupling for large data streams: The most convenient method with the highest performance is a direct cache coupling, whereas an AMBA based memory coupling will be sufficient for the beginning (to be discussed with ATAIR)
b) register coupling for small data and irregular MAC operations: Preferable is a direct coupling into the processors registers with an implicit synchronisation in the OF-stage of the processor pipeline. However coupling via load/store- or in/out-commands as external registers is acceptable- with the penalty of a higher latency which causes some performance limitation.

2 Specification of ALUr-PAE

2 .1 Overview

In a preferred embodiment, the ALU-PAE comprises 3 paths::
ALU arithmetic, logic and data flow handling

BP bypass

Then, each of the paths contains 2 data busses and 1 event bus. The busses of the DF path can be rerouted to the ALU path by configuration.

2.2 ALU path Registers

The ALU path comprises 12 data registers:

Ri0-3	Input data register 0-3 from bus
Rv0-3	Virtual output data register 0-3 to bus
Rd0-3	Internal general purpose register 0-3
Vi0-3	V event input 0-3 from bus
Ui0-3	U event input 0-3 from bus
Ev0-3	Virtual V event output register 0-3 to bus
Eu0-3	Virtual U event output register 0-3 to bus
Fu0-3	
Fv0-3	Internal Flag u and v registers according to the XPP-II PAE's event busses
Ace	Accumulator

Eight instruction registers are implemented, each of them is 24 bit wide according to the opcode format.

Rc0-7	Instruction registers

Three special purpose registers are implemented:

RIc	Loop Counter, configured by CM, not accessible through ALU-PAE itself. Will be decremented according to JL opcode. Is reloaded after value 0 is reached.
Rjb	Jump-Back register to define the number of used entries in Rc[0..7]. It is not accessible through ALU-PAE itself. If Rpp is equal to Rjb, Rpp is immediately reset to 0. The jump back can be bound to a condition i.e. an incoming event. If the condition is missing, the jump back will be delayed.
Rpp	Program pointer

2.3 Data duplication and multiple input reads

Since Function Folding can operate in a purely data stream mode as well as in a sequential mode (see 1.2) it is useful to support Ri reads in dataflow mode (single read only) and sequential mode (multiple read). The according protocols are described below:

Each input register Ri can be configured to work in one of two different modes:

Dataflow Mode
This is the standard protocol of the XPP-II implementation:
A data packet is taken read from the bus if the register is empty, an ACK handshake is generated. If the register is not empty ACK the data is not latched and ACK is not generated.
If the register contains data, it can be read once. Immediately with the read access the register is marked as empty. An empty register cannot be read.

Simplified the protocol is defined as follows:

```
     RDY & empty     → full
                     → ACK
     RDY & full            → notACK

     READ & empty    → stall
     READ & full           → read data
                     → empty
```

Please note: pipeline effects are not taken into account in this description and protocol.

Sequencer Mode
The input interface is according to the bus protocol definition: A data packet is taken read from the bus if the register is empty, an ACK handshake is generated. If the register is not empty ACK the data is not latched and ACK is not generated.
If the register contains data it can be read multiple times during a sequence. A sequence is defined from Rpp = 0 to Rpp = Rjb. During this time no new data can be written into the register. Simultaneously with the reset of Rpp to 0 the register content is cleared an new data is accepted from the bus.

Simplified the protocol is defined as follows:

```
     RDY & empty     → full
                     → ACK
     RDY & full            → notACK

     READ & empty    → stall
     READ & full           → read data

      (Rpp == Rjb)         → empty
```

Please note: pipeline effects are not taken into account in this description and protocol.

2.4 Data register and event handling

Data registers are directly addressed, each data register can be individually selected. Three address opcode form is used, $r_t \leftarrow r_{\varepsilon}\chi$, r_{s0}. An virtual output register is selected by adding 'o' behind the register. The result will be stored in r_t and copied to the virtual output register r_v as well according to the rule *op* out $(r_v, r_c) \leftarrow r_{si}$, r_{s0}.
Please note, accessing input and (virtual) output registers follow the rules defined in chapter 2.3.

Source	r_t	Notation
000	0	Rd0
001	1	Rd1
010	2	Rd2
011	3	Rd3
100	0	Ri0
101	1	Ri1
110	2	Ri2
111	3	Ri3

Target	r_t	r_v	Notation
000	0	-	Rd0
001	1	-	Rd1
010	2	-	Rd2
011	3	-	Rd3
100	0	0	Ro0
101	1	1	Ro1
110	2	2	Ro2
111	3	3	Ro3

Events are used equal to data registers. All input and internal events can be addressed directly, output events are used whenever an 'o' is added behind the event.

Etp	ep_t	ep_v	Notation
000	0	-	Fu0, Fv0
001	1	-	Fu1, Fv1
010	2	-	Fu2, Fv2
011	3	-	Fu3, Fv3
100	0	0	Eou0, Eov0

101	1	1	Eou1, Eov1
110	2	2	Eou2, Eov2
111	3	3	Eou3, Eov3

Es4/et4	e_t	e_v	Notation
0000	0	-	v0
0001	1	-	v1
0010	2	-	v2
0011	3	-	v3
0100	0	0	vo0
0101	1	1	vo1
0110	2	2	vo2
0111	3	3	vo3
1000	0	-	u0
1001	1	-	u1
1010	2	-	u2
1011	3	-	u3
1100	0	0	uo0
1101	1	1	uo1
1110	2	2	uo2
1111	3	3	uo3

2.4.1 Accumulator Mode

To achieve low power consumption and for better supporting DSP-like algorithms an accumulator register is available which can be addressed by just one set bit for the result register (ao) and operand register (ai).
For commutative operations always operand register 1 is replaced by ai. For non commutative operations as SUBtract operand register 1 selects, whether ai is the first or second operand. Operand register 2 defines the accordingly other operand.

It is to be noted that it has to he clarified whether a real Accumulator mode makes sense or just a MAC-command should be implemented to handle the multiply accumulate in a single command consuming two clock cycles with an implicit hidden accumulator access.

2.4.2 Parameter Stack Mode (PSTACK)

Unused entries in the Opcode Registers Rc can operate as stack for constants and parameters. At Rpp == 0000 the Rps PStack registers points to Rjb +1, which means the PStack area starts immediately behind the last entry in the Opcode register file.

To access the PStack, the FF-PAE must be in the Fast-Parameter Mode . Each read access to Ri3 is redirected to read from the PStack, whereas after each read access the pointer incremented with one. There is no check for an overflow of the PStack pointer implemented, an overflow is regarded as a program bug.

(see Fig. 28 now)

2.4.3 n:1 Transitions

n:1 transitions are not supported within the busses any more. Alternatively simple writes to multiple output registers Ro and event outputs Eo are supported. The Virtual Output registers (Rv) and Virtual Event (Ev) are translated to real Output registers (Ro) and real Events (Eo) , whereas a virtual register can be mapped to multiple output registers.

To achieve this a configurable translation table is implemented for both data registers and event registers:

Rv Ev	Ro0 Eo0	Ro1 Eo1	Ro2 Eo2	Ro3 Eo3
0				
1				
2				
3				

Example :
RvO mapped to RoO, RoI
RvI mapped to Ro2
Rv2 mapped to Ro3
Rv3 unused

Rv	Ro0	Ro1	Ro2	Ro3
0	1	1	0	0
1	0	0	1	0
2	0	0	0	1
3	0	0	0	0

2.4.4 Accessing input and output registers (Ri/Rv) and events (Ei/Ev)

Independently from the opcode accessing input or output registers or events is defined as follows:

Reading an input register:

Register status	Operation
empty	Wait for data
full	Read data and continue operation

Writing to an output register:

Register status	Operation
empty	Write data to register
full	Wait until register is cleared and can accept new data

2.4.5 Multi-Config Mode

The Multi-Config Mode allows for selecting 1 out of maximum 4 stored configurations. Incoming events on Fui θ,1 and FviO,1 select one of the 4 configurations. Only one Event shall be active at a clock cycle.

The selection is done by a simple translation, each event points to a specific memory address.

(see Fig. 29 now)

Long configurations may use more than 3 opcode by using the next code segments as well. In this case, the according events can not be used.

(see Fig. 30 now)

2.5 Opcode format

24 bit wide 3 address opcodes are used in a preferred embodiment :

$$\text{op } r_t \leftarrow r_a / \; Tb$$

Source registers can be Ri and Rd, target registers are Rv and Rd. A typical operation targets only Rd registers. If the source register for r_a is Ri [x] the target register will be Rd [x] .

The translation is shown is the following table::

Target	Source r_a
Rd0	Rd0
Rd1	Rd1
Rd2	Rd2
Rd3	Rd3
Rd0	Ri0
Rd1	Ri1
Rd2	Ri2
Rd3	Ri3

Each operation can target a Virtual Output Register Rv by adding an *out* tag 'o' as a target identifier to the opcode:

$$op\ (r_t,\ ro_t) \leftarrow r_a,\ r_b$$

Data is transferred to the virtual output register and to the according internal register as well:

Rv	Rd
Rv0	Rd0
Rv1	Rd1
Rv2	Rd2
Rv3	Rd3

2.5.1 Conditional Execution

The SKIPE command supports conditional execution. Either an event or ALU flag is tested for a specific value. Depending on the check either the next two addresses are executed (Rpp + 1) or skipped (Rpp + 3). If an incoming event is checked, the program execution stops until the event is arrived at the event port (RDY handshake set)..
SKIPE supports conditional execution of any OpCode which is not larger than two memory entries .
In SEQ-PAEs, which support CALL and RET OpCodes, also stack based subroutine calls are supported.

2.6 Clock

The PAE can operate at a configurable clock frequency of --
- 1x Bus Clock
- 2x Bus Clock
- 4x Bus Clock
- [8x Bus Clock]

2.7 The DF path

The DataFlow path comprises the data registers Brio..3 and Bro0..3 as well as the event register Bui/Bvi0..3 and Buo/Bvo0..3.

The main purpose of the DF path is to establish bus connections in the vertical direction. In addition the path includes a 4 stage FIFO for each of the data and event paths .

The DF path supports numerous instructions, whereas the in¬struction is selected by configuration and only one of them can be performed during a configuration, function folding is not available.

The following instructions are implemented in the DF path:

1. ADD, SUB
2. NOT, AND, OR, XOR
3. SHL, SHR, DSHL, DSHR, DSHRU
4. EQ, CMP, CMPU
5. MERGE, DEMUX, SWAP
6. SORT, SORTU
7. ELUT

2.9 Parameter Broadcast and Update

Parameters and constants can be updated fast and synchronous using input register Ri3 and event input Ei7 .

(see Fig. 31 now)

Depending on the update mode, data packets at the input register Ri3 are copied subsequently into Rd3, Rd2 and RdI at each access of the according register by the PAE, if the event Ei7 is set. Afterwards all input data at Ri3 is propagated to the output register Ro3 , also the Eo7 event output is set, to in¬dicate following PAEs the occurrence of a fast parameter up¬date, which allows to chain PAEs together (i.e. in a multi-TAP FIR filter) and updating all parameters in the chain.

Regis¬ter access	Ei7	UPM1 Upmcfg = 0100	UPM2 upmcfg = 1000	UPM3 upmcfg = 1100
-	0	-	-	-
read Rd3	1	Ri3 -> Rd3	Ri3 -> Rd3	Ri3 -> Rd3
read Rd2	1	Ri3 -> Ro3 1 -> Eo7	Ri3 -> Rd2	Ri3 -> Rd2
read	1	Ri3 -> Ro3	Ri3 -> Ro3	Ri3 -> RdI

Rd1		1 -> Eo7	1 -> Eo7	
-	1	Ri3 -> Ro3 1 -> Eo7	Ri3 -> Ro3 1 -> Eo7	Ri3 -> Ro3 1 -> Eo7

Also the OpCode UPDATE updates all registers subsequently if Ei7 is set, depending on the Update Parameter Mode (upmcfg = nn10) .
Also the register update can be configured to occur whenever Rpp == o and Ei7 is set by upmcfg = nn01.
In both cases nn indicates the number of registers to be updated (1-3) .

Ei7 must be 0 for at least one clock cycle to indicate the end of a running parameter update and the start of a new update.

3 Input Output Address Generators (IOAG)

The IOAGs are located in the RAM-PAEs and share the same registers to the busses. An IOAG comprises 3 counters with forwarded carries . The values of the counters and an immediate address input from the array are added to generate the address .
One counter offers reverse carry capabilities.

3 .1 Adressing modes

Several addressing modes are supported by the IOAG to support typical DSP-like addressing:

Mode	Description
Immediate	Address generated by the PAE array
xD counting	Multidimensional addressing using IOAG internal counters xD means ID, 2D, 3D
xD circular	Multidimensional addressing using IOAG internal counters, after overflow counters reload with base address
xD plus immediate	xD plus a value from the PAE array
Stack	decrement after "push" operations increment after "read" operations
Reverse carry	Reverse carry for applications such as FFT

3.1.1 Immediate Addressing

The address is generated in the array and directly fed through the adder to the- address- output . All counters are disabled and set to 0.

3.1.2 xD counting

Counters are enabled depending on the required dimension (x-dimensions require x counters) . For each counter a base address and the step width as well as the maximum address are configured. Each carry is forwarded to the next higher and enabled counter; after carry the counter is reloaded with the start address .
A carry at the highest enabled counter generates an event, counting stops.

3.1.3 xD circular

The operation is exactly the same as for xD counting, with the difference that a carry at the highest enabled counter generates an event, all counters are reloaded to their base address and continue counting.

3.1.4 Stack

One counter (CNTl) is used to decrement after data writes and increment after data reads . The base value of the counter can either be configured (base address) or loaded by the PAE array.

3.1.5 Reverse barry

Typically carry is forwarded from LSB to MSB. Forwarding the carry to the opposite direction (reverse carry) allows generating address patterns which are very well suited for applications like FFT and the like. The carry is discarded at MSB.

For using reverse carry a value larger than LSB must be added to the- actual value to count, wherefore the STEP register is used.

Example :
BASE = Oh
STEP = 1000b

Step	Counter Value

1	B0...00000
2	B0...01000
3	B0...00100
4	B0...01100
5	B0...00010
...	...
16	B0...01111
17	B0...00000

The counter is implemented to allow reverse carry at least for STEP values of -2, -1, +1, +2.

4. ALU/RAM Sequencers - SEQ-PAEs

Each ALU-PAE at the left or right edge of the array can be closely coupled to the neighbouring RAM-PAEs as an IP option, thus allowing for configure a sequencer. For compatibility reasons, .the data and opcode width of the sequencer is 1Sbita..

(see Fig. 32 now)

The ALU-PAEs can operate exactly as array internal ALU-PAEs but have several extensions. Operation is Sequencer mode the register file is 8 data registers wide,- Fu and Fv flags are used as carry, sign, null, overflow and parity ALU flag word.

Event Registers FF-Mode	Processor Registers SEQ-Mode
Fu0	carry
Fu1	sign
Fu2	null
Fu3	overflow
Fv0	parity

The address width is accordingly 16bit. However since the RAM-PAE size is limited it is segmented into 16 segments. Those segments are used for code, data and stack and must be individually preloaded by the compiler.

4 segment registers point to the specific segments:

CodeBank	Points to the actual code segment
DataBank	Points to the actual data segment
StackBank	Points to the actual stack segment
AuxiliaryBank	Points to any segment (but code), allowing copy operations between segments

(see Fig. 33 now)

The compiler has to take- care that necessary data segments are preloaded and available. For cost reasons there is no automatic TLB installed.
Also segments have to be physically direct addressed due to the absence of TLBs. This means that the compiler has to implement range checking functions for according addresses.

Code segments behave accordingly to data segments. The compiler has to preload them before execution jumps into them. Also jumps are physically direct addressed, due to the absence of TLBs again.

A relocation of any segments is not possible, the mapping is fixed by the compiler.

The memory layout is shown in Fig. 34 now. Here, a simple check mechanism is implemented to validate or invalidate memory segments.

At least the CodeBank (CB) and StackBank (SB) must be set. The first CodeBank must start at location 0000h. For all other banks 0000h is an illegal entry. Loading segments to the memory validates them, accordingly flushing invalidates them.

Memory banks are updates in terms of loaded or flushed in the background by a DMA engine controlled by the following opcodes

LOADDSEG	Loads and validates a data/auxiliary/stack bank
STOREDSEG	Stores and invalidates a data/ auxiliary/ stack bank
LOADCSEG	Loads and validates a code bank

The address generators in the IOAG interfaces can be reused as DMA engine .

Memory banks can be specifically validated or invalidated as follows :

VALIDATESSEG	Validates a bank
INVALIDATESEG	Invalidates a bank

The bank pointers are added to the address of any memory access. Since the address pointer can be larger than the 6 bits addressing a 64 line range, segment boarders are not "sharp", which means, can be crossed without any limitation. However the programmer or compiler has to take care that no damage occurs while crossing them. If an invalid segment is reached a flag or trap is generated indicating the fault, eventually just wait states are inserted if a segment preload is running already in the background.

(see Fig. 35 now)

Alternatively a more advanced valid checking scheme can be implemented as shown in Fig. 36 now.

In difference to PAEs which require 24-bit instructions sequencers use 16-bit instructions only. To use the same instruction set and to keep the decoders simple, just the last 8 bits are discarded in sequencer mode.

4.1 IOAGs

IOAGs may comprise a 4-8 stage data output buffer to balance external latency and allow reading the same data address directly after the data has been written, regardless of external bus or memory latencies (up to the number of buffer stages).

In the follwoing, a number of OpCodes and their meanings is suggested:

ADD

ADD

Description:
Add rsl and rs2.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input 12 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo -

et4	
0nnn	F[nnn], F[nnn]

1nnn	Eo [nnn] , Eo [nnn]

IX, I2- -> O
Rpp++

rs : source register
rt : target register
et4 : target event

Input Registers :
Ri / Rd

Output Registers :
Rd / Ro

Input Flags:
F , Ei

Output Flags :

Mode	
SEQ	carry, sign, null, parity
FF	carry -> Fu / Euo

ADDC

ADD with Carry

Description:
Add rs1 and rs2 with Carry.

Action:
Input I1 =

rs1	
0nn	Rd [nn]
1nn	Ri [nn]

Input I2 =

rs2	
0nn	Rd [nn]
1nn	Ri [nn]

Event Input E=

es4	
0nnn	F [nnn]
1nnn	Ei [nnn]

Output O =

rt	

0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

etp	
0nn	Fu[nn], Fv[nn]
1nn	Euo[nn], Evo[nn]

II, 12 -> O
Rpp++

rs :	source register
rt :	target register
es4 :	source event
etp:	target event pair

Input Registers :
Ri / Rd

Output Registers:
Rd / Ro

Input Flags :
F, Ei

Output Flags :

Mode	
SEQ	carry, sign, null, parity, over-flow
FF	carry -> Fu / Euo, overflow -> Fv / Eyo

AND

Logical AND

Description:
Logical AND operation

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input 12 =

Irs2	I

0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

et4	
0nnn	F[nnn], F[nnn]
1nnn	Eo[nnn], Eo[nnn]

II, 12 -> O
Rpp++

rs : sourqe register
rt : target register

Input Registers :
Ri / Rd

Output Registers :
Rd / Ro

Input Flags :
-

Output Flags :

Mode	
SEQ	zero, sign , parity
FF	zero, sign -> F / Eo

BSHL

Barrel SHift Left

Description :
Shift rsl left by rs2 positions and fill with zeros,.

Action:
Input I1 =

rs1	
0nn	Rd[nn]

Inn	TRi [nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

II, I2 -> O
Rpp++

rs: source register
rtp: target register pair

Input Registers:
Ri / Rd

Output Registers:
Rd / Ro

Input Flags :
-

Output Flags :
-

BSHR

Barrel SHift Right

Description:
Shift rs1 right by rs2 positions, sign bit is duplicated.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

II, 12 -> O
Rpp++

rs: source register
rtp: target register pair

Input Registers:
Ri / Rd

Output Registers::
Rd / Ro

Input Flags :
-

Output Flags :
-

BSHRU

Barrel SHift Right Unsigned

Description:
Shift rs1 right by rs2 positions and fill with zeros.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input 12 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output 0 =

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)],

	Ro [(n*2)+1]

I1, I2 -> O
RPP++.

rs : source register
rtp : target register pair

Input Registers :
Ri / Rd

Output Registers :
Rd / Ro

Input Flags :
-

Output Flags :
-

CLZ

Count Leading Zeros

Description:
Count the amount of leading zeros if the number is positive, accordingly, count the amount of leading ones if the number is negative.

Action:
Input I1 =

rs1	
0nn	Rd [nn]
1nn	Ri [nn]

Output 0 =

rt	
0nn	Rd [nn]
1nn	Ro [nn]

Event output Eo =

etp	
0nn	Fu[nn], Fv[nn]
1nn	Euo[nn], Evo[nn]

I1 -> 0
Rpp++

rs/:	source register
ft:	target register
etp:	target event pair

Input Registers :
Ri / Rd

Output Registers:
Rd / Ro

Input Flags :
-

Output Flags :

Mode	
SEQ	sign, parity, zero
FF	sign, zero -> F / Eo

CLZU

Count Leading Zeros Unsigned

Description:
Count the amount of leading zeros of an unsigned number.

Action:

Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Output 0 =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

et4	
0nnn	F[nnn]
1nnm	Eo[nnn]

I1 -> O
Rpp++

rs:	source register
rt:	target register

et4 : target event

Input Registers :
Ri / Rd -

Output Registers:
Rd / Ro

Input Flags :
-

Output Flags :

Mode	
SEQ	sign, parity, zero
FF	zero -> F / Eo

CMP

CoMPare

Description:
Compare two values

Action:
Input I1 =

rs1	
0nn	Rd [nn]
1nn	Ri [nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Event output Eo =

etp	
0nn	Fu[nn], Fv[nn]
1nn	Euo[nn], Evo[nn]

Rpp++

rs : source register
etp: target event pair

Input Registers:
Ri / Rd

Output .Registers:

-

Input 'Flags :

-

Output Flags :

Mode	
SEQ	sign, zero
FF	sign, zero -> F / Eo

CMPU

CoMPare Unsigned

Description:

Compare two unsigned values,.

Action:

Input I1 =

rs1	
0nn	Rd [nn]
1nn	Ri [nn]

Input 12 =

rs2	
0nn	Rd [nn]
1nn	Ri [nn]

Event output Eo =

etp	
0nn	Fu [nn], Fv [nn]
1nn	Euo [nn], Evo [nn]

R-PP++

rs:	source register
etp :	target event pair

Input Registers :

Ri / Rd

Output Registers :

-

Input/ Flags :

-

Output Flags:

Mode	
SEQ	sign, zero
FF	sign, zero -> F / Eo

DEMUX **FF**

DEMUltipleX data stream

Description:
Moves input to one of two outputs, depending on flag.

Action:
Input I -

rs	
0nn	Rd [nn]
1nn	Ri [nn]

Output 01 =

rt1	
0nn	Rd [nn]
1nn	Ro [nn]

Output 02 =

rt2	
0nn	Rd [nn]
1nn	Ro [nn]

Event E=

es4	
0nnn	F [nnn]
1nnn	Ei [nnn]

E	
0	O1 = I
1	O2 = I

Rpp++

rt : target register
rs : source register
es4 : source event

Input Registers:
Ri / Rd

Output Registers :
Rd / Ro, Rd / Ro

Input Flags :
Ei / F

Output Flags:
-

DIV **SEQ**

DIVide

Description:
Divide rs1 by rs2 . Result in rtp, reminder in rtp+1 .

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

II, I2 -> O
Rpp++

rs : source register
rtp: target register pair

Input Registers:
Ri / Rd

Output Registers:

Rd / Ro

Input Flags :

-

Output Flags :

-

DIVU SEQ

Divide Unsigned

Description:

Divide unsigned rsl by rs2. Result in rtp, reminder in rtp+1 .

Action:

Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input 12 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

OutPut O =

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

II, 12 -> O

_Rpp++

rs : source register
rtp: target register pair

Input Registers:

Ri / Rd

Output Registers; :

Rd / Ro

Input Flags :

-

Output Flags :

DSHL

Double SHift Left

Description:
Shift rsl and rs2 left. LSB is filled with event,.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input 12 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Event Input E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

Output O =

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

Event output Eo =

etp	
0nn	Fu[nn], Fv[nn]
1nn	Euo[nn], Evo[nn]

II, 12 -> O
Rpp++

rs:	source register
rtp:	target register pair
etp:	target event pair

Input Registers:
Ri / Rd

Output Registers ;
Rd / Ro

Input Flags:
F, Ei

Output Flags :

Mode	
SEQ	MSB(rs1) -> carry, MSB(rs2) -> sign
FF	MSB(rs1) -> Fu / Euo, MSB(rs2) -> Fv / Evo

DSHR

Double SHift Right

Description:
Shift rsl and rs2 right, sign bit is duplicated..

Action:
Input I1 =

rsl	
0nn	Rd[nn]
1nn	Ri[nn]

Input 12 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output 0 =

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

Event output Eo =

etp	
0nn	Fu[nn], Fv[nn]
1nn	Euo[nn], Evo[nn]

II, 12 -> 0
Rpp++

rs: source register
rtp: target register pair

etp: target event pair

Input Registers :
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
Ei, F

Output Flags :

Mode	
SEQ	LSB(rs1) -> carry, LSB(rs2) -> sign
FF	LSB(rs1) -> Fu / Euo, LSB(rs2) -> Fv / Evo

DSHRU

Double SHift Right Unsigned

Description:
Shift rs1 and rs2 right and fill with event.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input 12 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Event Input E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

Output Q =

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

Event/output Eo =

etp	
0nn	Fu[nn], Fv[nn]
1nn	Euo[nn], Evo[nn]

I1, I2 -> O
Rpp++

rs:	source register
rtp:	target register pair
etp:	target event pair

Input Registers :
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
Ei, F

Output Flags:

Mode	
SEQ	LSB(rs1) -> carry, LSB(rs2) -> sign
FF	LSB(rs1) -> Fu / Euo, LSB(rs2) -> Fv / Evo

EQ

EQual

Description:
Check whether two values are equal.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Event output Eo =

et4	
0nnn	F[nnn]
1nnn	Eo[nnn]

Rpp++

rs : source register
et4 : target event

Input Registers :
Ri / Rd

Output Registers :
-

Input Flags:
-

Output Flags :

Mode	
SEQ	zero
FF	zero -> F / Eo

JMP **SEQ**

JuMP immediate

Description:
Jump to address defined by immediate constant. CodeBank is changed according to constant .

Action:
const [0..3] -> CodeBank
const [4..15] -> Rpp

Input Registers :
-

Output Registers :
-

Input Flags :
-

Output Flags:
-

JRI SEQ

Jump Relative Immediate

Description:
Jump relative to Rpp according to immediate signed constant.
CodeBank is not influenced.

Action:
Rpp + const -> Rpp

Input Registers :
-

Output Registers :
-

Input Flags :
-

Output Flags :
-

JRR SEQ

Jump Relative Register

Description:
Jump relative to Rpp according to signed content of register.
CodeBank is not influenced.

Action:
Rpp + Rd[rbs] -> Rpp

Input Registers;
-

Output Registers :
-

Input Flags:
-

Output Flags :
-

LOAD

LOAD data register with constant

Description:
Loads internal data register or output register with an immediate constant

Action:

rt	
0nn	const -> Rd[nn]
1nn	const -> Ro[nn]

Rpp++

rt: target register

Input Registers :
-

Output Registers::
Rd /Ro

Input Flags:
-

Output Flags :
-

MERGE FP

MERGE data streams

Description :
Moves one of two inputs to output, depending on flag..

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input 12 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output Q =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event: E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

E	
0	O = I1
1	O = I2

Rpp++

rt :	target register
rs :	source register
es :	source event

Input Registers:
Ri / Rd, Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
Ei / F

Output Flags:
-

MOVE

MOVE internal data register

Description:
Moves content of a register bank register to another internal register.

Action:
Rd [rbs] -> rd [rbt]
Rpp++

rbs:	register bank source
rbt:	register bank target

Input Registers :
Rd

Output Registers :

Rd

Input Flags :

-

Output Flags:

-

MOVEE

MOVE flag register

Description:
Moves content of a flag register to another flag register..

Action:
F[fs] -> F [ft]
Rpp++

fs : flag source
ft: flag target

Input Registers :
-

Output Registers ::
-

Input Flags:
F

Output Flags ::
F

MUL

MULtiply

Description:
Multiply rs1 and rs2.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output. O =

rtp	
0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

II, 12 -> O

R-PP+ +

rs : source register

rtp: target register pair

Input Registers :

Ri / Rd

Output Registers: :

Rd / Ro

Input Flags:

-

Output Flags :

-

MULU

MULtiply Unsigned

Description:

Multiply unsigned rsl and rs2 .

Action:

Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input 12 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rtp	

0n	Rd[(n*2)], Rd[(n*2)+1]
1n	Ro[(n*2)], Ro[(n*2)+1]

Ii, I2 -> O
Rpp++

rs: source register
rtp: target register pair

Input Registers :
Ri / Rd

Output Registers::
Rd / Ro

Input Flags :
-

Output Flags :
-

NOP

No Operation

Description;
No Operation, Rpp is incremented

Action:
Rpp++

Input Registers:
-

Output Registers:
-

Input Flags :
-

Output Flags :
-

NOT

Logical inverse

Description:
Inverts register logically

Action:
Input I =

rs	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

I -> 0
Rpp++

rs::	source register
rt;:	target register

Input Registers:
Ri / Rd

Output Registers :
Rd / Ro

Input Flags :
-

Output Flags:

Mode	
SEQ	zero
FF	F / Eo

OR

Logical OR

Description:
Logical OR operation

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input 12 =

rs2	

0nn	Rd[nn]
1nn	Ri[nn]

Output. O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

II, 12 -> O
Rpp++

rs : source register
rt : target register

Input Registers :
Ri / Rd

Output Registers :
Rd / Ro

Input Flags :
-

Output Flags:

Mode	
SEQ	zero, sign , parity
FF	zero -> F / Eo

READ

READ data input register

Description:
Read specified data input register and write to internal register bank or output register. READ waits until data is available at the input register.

Action:

rt	
0nn	Ri[ri] -> Rd[nn]
1nn	Ri[ri] -> Ro[nn]

Rpp++

rt : target register
ri : input register

Input Registers:
Ri

Output Registers:
Rd / Ro

Input Flags:
-

Output Flags :
-

READE

READ event input register

Description:
Read specified event input register and write to internal flag bank or event output register. READE waits until event is available at the input register.

Action:

et4	
0nnn	Ei[ei] -> F[nnn]
1nnn	Ei[ei] -> Eo[nnn]

Rpp++

et4 : target event
ei : input event

Input Registers:
-

Output Registers :
-

Input Flags :
Ei

Output Flags :
F / Eo

SAT

SATurate

Description:
Saturates register depending on carry (FuO) flag and saturation mode.

Action:
Input I =

rs	
0nn	Rd[nn]
1nn	Ri[nn]

Output Q =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event E=

Mode	es4	
SEQ	don't care	carry
FF	0nnn	F[nnn]
FF	1nnn	Ei[nnn]

E	as	
0	don't care	I -> O
1	0	0h -> O
1	1	ffffh -> O

Rpp+ +

rs:	source register
rt:	target register
as:	add/ substract mode
es4:	event source

Input Registers :
Rd

Output Registers:
Rd / Ro

es4Input Flags:
SEQ-Mode : carry-
FF-Mode: Ei/F

Output Flags:
-

SETF

SET Flag with constant

Description:
Loads flag register or output event with an immediate constant

Action:

et4	
0nnn	const -> F[nnn]
1nnn	const -> Eo[nnn]

Rpp++

et4: event target

Input Registers:
-

Output Registers:
-

Input Flags:
-

Output Flags:
F /Eo

SHL

SHift Left

Description:
Shift rsl left. LSB is filled with event.

Action:
Input I1 =

rsl	
0nn	Rd[nn]
1nn	Ri[nn]

Event Input E =

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

et4	
0nnn	F[nnn]
1nnn	Eo[nnn]

I1 -> O
Rpp++

rs :	source register
rt :	target register pair
et4 :	target event pair
es4 :	source event register

Input Registers :
Ri / Rd

Output Registers :
Rd / Ro

Input Flags :
F, Ei

Output Flags ;

Mode	
SEQ	MSB(rs1) -> carry
FF	MSB(rs1) -> Fu / Euo

SHR

SHift Right

Description :
Shift rsl right. MSB is filled with event.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Event Input E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

et4	
0nnn	F[nnn]
1nnn	Eo[nnn]

```
I1 -> O
Rpp++
```

rs :	source register
rt :	target register pair
et4 :	target event pair
es4 :	source event register

Input Registers :
Ri / Rd

Output Registers:
Rd / Ro

Input Flags :
F, Ei

Output Flags :

Mode	
SEQ	LSB(rs1) -> carry
FF	LSB(rs1) -> Fu / Euo

SKIPE

SKIP next two commands depending on Event

Description:
Next two commands are skipped based on event or flag. If an event is selected as source the execution stops until the event is available.

Action:

val	value
0	0
1	1

Event E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

Skip next two addresses if event or flag is equal to val:

event/flag	
not equal val	Rpp++
equal val	Rpp + 3 -> Rpp

val :	value
es4 :	event source

Input Registers:

-

Output Registers :

-

Input Flags:

Ei / F

Output Flags :

-

SORT **PF**

SORT data stream

Description:

Sort two inputs, depending on value,.

Action:

Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]

Inn	Ri [nn]

Output O1 =

rt1	
0nn	Rd [nn]
1nn	Ro [nn]

Output 02 =

rt2	
0nn	Rd [nn]
1nn	Ro [nn]

Event E1=

et41	
0nnn	F [nnn]
1nnn	Eo [nnn]

Event E2=

et42	
0nnn	F [nnn]
1nnn	Eo [nnn]

01 = smaller value of I1 and 12
02 = larger value of I1 and 12

E1 = 1 if I1 < 12 else 0
E2 = 1 if I1 <= 12 else 0

Rpp++

rt :	target register
rs :	source register
et4 :	target event .

Input Registers :
Ri / Rd, Ri / Rd

Output Registers:
Rd / Ro, Rd / Ro

Input Flags :
-

Output Flags :
Ei / F

SORTU FF

SORT data stream Unsigned

Description:
Sort two unsigned inputs, depending on value.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input 12 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output 01 =

rt1	
0nn	Rd[nn]
1nn	Ro[nn]

Output 02 =

rt2	
0nn	Rd[nn]
1nn	Ro[nn]

Event E1=

et41	
0nnn	F[nnn]
1nnn	Eo[nnn]

Event E2==

et42	
0nnn	F[nnn]
1nnn	Eo[nnn]

01 = smaller value of I1 and 12
02 = larger value of I1 and 12

E1 = 1 if I1 < 12 else 0
E2 = 1 if I1 <= 12 else 0

Rpp++

rt : target register
rs : source register

et4: target event

Input Registers :
Ri / Rd, Ri / Rd

Output Registers :
Rd / Ro, Rd / Ro

Input Flags :
-

Output Flags:
Ei / F

SUB

SUBtract

Description:
Subtract rs2 from rs1..

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

et4	
0nnn	F[nnn], F[nnn]
1nnn	Eo[nnn], Eo[nnn]

II, 12 -> O
Rpp++

rs: source register
rt: target register

et4: target event

Input Registers :
Ri / Rd

Output Registers:
Rd / Ro

Input Flags:
F, Ei

Output Flags:

Mode	
SEQ	carry, sign, null, parity
FF	carry -> Fu / Euo

ADDC

ADD with Carry

Description:
Subtract rs2 from rs1 with Carry.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input I2 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Event Input E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

Output O =

rt	
0nn	Rd[nn]
1nn	Ro[nn]

Event output Eo =

etp	
0nn	Fu[nn], Fv[nn]

1nn	Euo[nn], Evo[nn]

I1, I2 -> O
Rpp++

rs: source register
rt: target register
es4 : source event
etp: target event pair

Input Registers:
Ri / Rd

Output Registers :
Rd / Ro

Input Flags :
F, Ei

Output Flags :

Mode	
SEQ	carry, sign, null, parity, over-flow
FF	carry -> Fu / Euo, overflow -> Fv / Evo

SWAP **FF**

SWAP data stream

Description:
Swap two inputs, depending on flag.

Action:
Input I1 =

rs1	
0nn	Rd[nn]
1nn	Ri[nn]

Input 12 =

rs2	
0nn	Rd[nn]
1nn	Ri[nn]

Output 01 =

rt1	
0nn	Rd[nn]
1nn	Ro[nn]

Output 02. =

rt2	
0nn	Rd[nn]
1nn	Ro[nn]

Event E=

es4	
0nnn	F[nnn]
1nnn	Ei[nnn]

E	
0	01 = 11, 02 = 12
1	01 = 12, 02 = I1

Rpp++

rt : target register
rs : source register
es4 : source event

Input Registers :
Ri / Rd, Ri / Rd

Output Registers :
Rd / Ro, Rd / Ro

Input Flags :
Ei / F

Output Flags :
-

UPDATE **FF**

UPDATE parameters

Description :
Updates registers Rd3, Rd2 , RdI with value from Ri3 if Ei7 is set. Moves subsequent data packet on Ri3 to Ro3 and sets Eo7 .

Action:

Mode				
1	Ri3 -> Rd3	set Eo7 Ri3 -> Ro3		
2	Ri3 -> Rd3	Ri2 -> Rd2	set Eo7 Ri3 -> Ro3	

3	Ri3 -> Rd3	Ri2 -> Rd2	RiI -> RdI	set Eo7 Ri3 -> Ro3

Rpp++

'rnode : update mode

Input Registers :
Ri3

Output Registers :
Rd3, Rd2, RdI

Input Flags :
Ei7

Output Flags:
Eo7

WAITE

WAIT for incoming Event

Description:
Stop execution and wait for incoming event of defined value . Acknowledge incoming events.

Action:

valx	value
00	0
01	1
1x	don't care

Event E =

es3	
nnn	Ei [nnn]

Wait for incoming event of defined value. Acknowledge all incoming events..

valx: value
es3 : event source

R-PP+ +

Input Registers :
-

Output Registers :
-

Input Flags :
Ei

Output Flags :
-

WRITE

WRITE output register

Description:
Write data from input register or internal register bank to output register. Wait for incoming ACK either before or after writing.

Action:
<sync θ>

rs	
0nn	Ri[nn] -> Ro[ro]
1nn	Rd[nn] -> Ro[ro]

<sync1>
Rpp++

ro:	output register
rs:	register source

Synchronisation is handled according to sy:

sy = 0 <sync θ>	Wait only if previously sent event has not been granted by ACK yet
sy = 1 <synσ1>	Wait until actual event is granted by ACK

Input Registers :
Ri / Rd

Output Registers :
Ro

Input Flags:
-

Output Flags.:

WRITEIS

WRITE Event output register

Description:
Write event from input register or flag to event output register. Wait for incoming ACK either before or after writing.

Action:
<sync θ>

es4	
0nnn	Ei[nnn] -> Eo[eo]
1nnn	F[nnn] -> Eo[eo]

<sync1>
Rpp++

eo : output event
es4 event source

Synchronisation is handled according to sy :

sy = 0 <sync θ >	Wait only if previously sent event has not been granted by ACK yet
sy = 1 <sync1 >	Wait until actual event is granted by ACK

Input Registers:
-

Output Registers :
-

Input Flags :
Ei / F

Output Flags:
Eo

XOR

Logical XOR

Description:

Logical XOR operation

Action:

input I1 =

rsl	
Onn	Rd [nn]
inn	Ri [nn]

Input 12 =

rs2	
Onn	Rd tnn]
Inn	Ri [nn]

Output O =

rt	
Onn	Rd [nn]
Inn	Ro [nn]

I I , 12 -> O
Rpp++

rs :	source register
rt :	target register

Input Registers :
Ri / Rd

Output Registers:
Rd / Ro

Input Flags :
-

Output Flags :

Mode	
SEQ	zero, sign , parity
FF	zero -> F / Eo

Appendix B

In the following, an exaple for the use of function folding is given:

Function Folding and Fast Parameter Update Example FIR

RiO = x
RiI = y

3-folded FIR using ace
Fast parameter- update for registers RdI , Rd2 , Rd3

<u>example 1 ; UPM3, updates parameters with each access to Rd3,2,1 (if Ei7 is set)</u>

```
upmcfg = 1100
# stage 1
     tnul ace, RiO, Rd3 ;
     add RdO, ace, RiI;
# stage 2
     mul ace, RiO, Rd2;
     add RdO, ace, RdO;
# stage 3
     mul ace, RiO, RdI;
     add RoI, ace, Rd3 ;
     write RoO, RiO;
```

Alternative using MAC opcode, parameter pop and looping

```
           read RdO, RiI;
Ih, It [3]:      mac RdO , RiO, pop;
                 write RoI, RdO;
           write RoO, RiO;
```

<u>example 2 ; UPM3 , uses command UPDATE for parameter update</u>

```
upmcfg = 1120
# stage 1
     mul ace, RiO, Rd3;
     add RdO, ace, RiI;
# stage 2
     mul ace, RiO, Rd2 ;
     add RdO, ace, RdO;
# stage 3
     mul ace, RiO , RdI;
     add RoI, ace, Rd3 ;
     write RoO , RiO;
     update 3
```

<u>example 3 : UPM3, updates parameters at Rpp == 0</u>

```
upmcfg = 1101
# stage 1
     mul ace, RiO , Rd3 ;
     add RdO, ace, RiI;
# stage 2
     mul ace, RiO , Rd2 ;
     add RdO , ace, RdO;
# stage 3
     mul ace, RiO, RdI;
     add RoI, ace, Rd3;
     write RoO, RiO;
```

In the above, an improved data processor array has been described. Although only in some instances, it has been pointed out that reference to a certain number of registers, bit width etc. is for explanation only, it is to be understood that this also holds where such reference is not found.

If the array is to be very large or in case a real time process is run where two different fragments of an array unknown at compile time have to communicate with each other so as to enable data processing, it is advantageous to improve the performance by ensuring that a communication path can be set up. Several suggestions have been made already, e.g. Lee-Routing and/or the method described in PACT 7. It is to be understood that the following part of an improved array design might result in an improved circuitry for certain applications but that it is not deemed absolutely and inevitably necessary to implement it with e.g. a function fold PAE. Rather, the other suggestions for improvement will result in significant improvements on their own as will be understood by the average skilled person.

ROUTING IMPROVEMENT

The suggested improvement described hereinafter concerns the static routing network for reconfigurable array architectures. Hereby this static network is enhanced by implementing additional logic to adaptive runtime routing.

Figure 1 depicts a cut-out of a reconfigurable array with a set of functional units (PU). Each functional unit encloses one routing unit (RU) and additional functional modules (FMs). The enclosed functional modules are used to manipulate data and characterize the type of the FU. The RU contains an interconnect matrix which is able to route each input port to any desirable output ports. All FUs are connected through point-to-point links whereas each is composed of two half-duplex links and able to transport the data in both directions at the same time.

The routing technique described in this document is instruction based which means that each routing process must be started by an instruction. If the user wants to establish a routing between two cells, he has to bring a specific instruction into the source cell. The hardware within the array calculates based on the instruction fields values the desired routing direction and establishes the logic stream. The routing process happens stepwise from one functional unit to another whereby each cell decides which direction should be taken next. On the way to an established route we defined three valuable states of the routing resources. The first state is the physical route or link. This means that the resources of this route are not used and available to routing processes. The second state is named temporal route or link. This state describes the temporarily not available link, which means that this link is in use for routing purposes- but the mentioned routing is not confirmed yet. The problem here is that this route can be confirmed in the future or released if the successor cells are able to re-alise the desired routing. The last state is the logical route or link. This state represents an established route on the array which is able to transport calculation data.

This routing technique uses coordinates on the array to calculation routings. Each FU possesses unique coordinate's und on the basis of this information it is able to determine the routing direction to each desired cell within the array. This concept is the basis for the adaptive runtime routing described in this document. The needed control logic for adaptive routing is implemented within the routing unit, especially within the rout-ing controller which controls the interconnect matrix at runtime. Therefore the routing controller is able to analyze the incoming data of all input ports of the concerned FU and come to a decision what to do next.

Routing Establishment

For the purpose of incoming data analyzing and data buffering each input port owns so called in-registers (InReg). Additional to those standard reg-isters there are InReg-controllers implemented (InRegCtrl). Those finite state machines (FSMs) have the job to store the actual state of the input links and in dependency of the actual state to trigger routing requests or release not required routings. To fulfil its job each InRegCtrl is con-nected to an in-controller (InCtrl) which is implemented exactly once per FU. Important requirement for requesting of new routings is that the men-tioned input resource (InReg, InRegCtrl) are not used and so in the state of physical link.
InCtrl gets requests of all InRegCtrls all over the time and forwards one request after another to the routing controller (RoutCtrl). The selection which InRegCtrl should be served first is dependant on the routing priority of the input link and/or which input link was served last. Based on trie co-ordinate information of the target cell and the coordinates of the actual FU the RoutCtrl calculates the forward direction for the requested input link. Thereby the RoutCtrl takes into account additional parameters like optimum bit (will be described later), the network utilisation towards the desired direction, etc.
If the direction calculation within the RoutCtrl was successful the RoutCtrl forwards the request with additional information about the output port to the interconnect matrix, which connects the input port with calcu-lated output port. If this is done the RoutCtrl signals the successful routing operation to InCtrl. Because the actual reached routing state is not final it is necessary to store the actual state. This happens within the queue-request-registerfile (QueueRRF). Therefore the InCtrl is directly connected to the QueueRRF and is able to store the desired information. At this point the related input and output links reach the temporal link state and are temporarily not available for other routing processes.
Due the fact that the QueueRRF is able to store more than one routing en-try, the InCtrl is able to hold multiple routing processes at the same time. But for the purpose of high hardware area consumption the direction calculation is realized once within the RoutCtrl.
The established temporal routing stays stored within the QueueRRF till the point the successor cell acknowledges the routing. In this case the InCtrl clear the according entry in the QueueRRF and signals the successful rout-ing to the InCtrl. The InRegCtrl changes into the state logical route and signal the predecessor cell the successfully finished routing process.
The other case can happen if the successor cell is not able to establish the desired route. In this case the InCtrl forwards a new request to the RoutCtrl based on the QueueRRF-entry. This request leads to new routing suggestion which will be stored within the QueueRRF.

If all available and expedient directions are checked and routing trials failed the InCtrl signals to InRegCtrl the failed routing. The InCtrl signals the same routing miss to the predecessor cell and finishes the routing process in the current cell.
Within the routing process there are two exceptions how the routing unit establishes a desired routing. Those exceptions affect the source and the target cell. The exception in both cases is that as well the source cell as the target cell do not need to route the started/ending routing through the interconnect matrix. To connect the FMs to the output links of cells simple multiplexers are used. Those multiplexers are implemented after the interconnect matrix and have to be switched explicitly. This happens after the routing process is finished. The exception lies in the finishing state. Here the InRegCtrl doesn't have to acknowledge the successful routing the predecessor it just has to consume the actual routing instruction in the InReg instead. This happens after the InCtrl signals the successful routing. Additionally the InReg switches the output multiplexer associated to the output port of the FM and finishes the routing establishment. The information needed the switch the right output multiplexer gets the InCtrl from the RoutCtrl .
Otherwise if the routing fails the InCtrl asserts cell specific interrupt line and signals the failure to the system.
The second exception concerns the target routing cell. Here it is important to connect the new route with the input ports of the local FM. Therefore simple multiplexers are used which are implemented before the interconnect matrix. If an ongoing routing process reaches the target cell the InCtrl identifies the target achievement and switches the associated input multiplexer to forward the incoming data to the input port of the FM. This is the point where the successful route establishment signal is generated by the InRegCtrl after InCtrl signals the success. Here the InRegCtrl has the last job to finish the routing process by deleting the routing instruction and going to logical state.

Releasing Established Routing

For releasing of the logically established routings we introduced special instructions, so called end packets. The only purpose of those instructions is the route-dissolving by inject the necessary end packet into the logic established routing. There are two ways how the routings can be released. The first possibility is the global releasing. This means that all routes which are following the route where the end packet is injected will be released. This function is useful to delete whole configurations with one single instruction. For this purpose it is important that the FMs are able to forward the end packet unaltered through the internal datapaths.
The second way for route releasing is the local route releasing. Here it is possible to release single established routes between output and input ports of FMs . The end packets are not propagated through the FMs . In this case the end packet will be consumed by the last InRegCtrl.
The internal RU communication is similar to the routing process. If the InRegCtrl determines incoming end packet and the InRegCtrl is in the logic route state, the InRegCtrl forwards the route release request to the InCtrl. The InCtrl clears the entries either within the interconnect matrix or within the input multiplexers registers or within the output multiplexer registers. Meanwhile the InRegCtrl consumes (in case of the local end packet and last cell in the chain) the instruction and goes to the idle state. If the end packet was a global instruction the InRegCtrl forwards alway the end packet to the successor.

-Additional Features

For the purpose of priority control, we introduced a priority system to influence the order in which the RU serves the incoming routing requests. Therefore the instructions contain priority fields which describe the priority level. Higher values in this field result in higher priority und will be preferred by the RU during the runtime routing. The priority field has

direct influence on the selection of the incoming routing requests from the InRegCtrls to inCtrl.
Some inner configuration communication streams require strictly defined latency to reach the desired performance. Therefore it is very important to keep the maximum register chain length. To decrease the latency of the routed' streams its is necessary to ensure that the array chose always the best routing between source and target, but this requirement may lead to not routable streams if this feature will be always required. To ease this problem we introduced a special bit within the routing instruction, so called optimum bit (OptBit) . This bit has to be activated if the optimum routing is definitely required. In this case the array tries to reach this requirement und delivers an interrupt if fails.
The alternative to reach the required latency is the speed path counter. This counter gives the possibility to bypass a specific number of registers before buffering again. Therefore we defined a reference value and the counter value. Both numbers are stored within the instruction field. Each passed cell respective the RU compares the counter value and the reference value. If both values are equal then the actual cell buffers the stream and resets the counter. If the counter is smaller than the reference value the current buffer will be bypassed and the counter incremented by one. In this way it is possible to bypass a number of buffers which equals exactly to reference value.

MuI ti-grained Communication Links

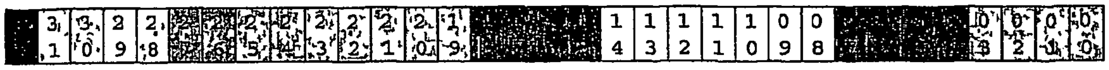
In addition to the coarse-grained point-to-point links we introduced more flexible multi-grained point-to-point links. Hereby one single point-to-point link connects two neighbor cells respective the RUs within those cells. One coarse-grained link consists of a set of wires, e.g. 32 wires for one 32 link, and additionally protocol signals. The whole vector is handled by a single set of control signals which makes this communication resource not usable for multi -grained communication.
To reach this requirement we divided the whole 32 bit vector into single strips, e.g. with groups of 8 times 1 bit strips and 3 times 8 bit strips. Each strip obtained separate control signals and is able to operate independently from other strips.
The idea behind this division is to combine those strips to logical multi-grained sub-links. If you have one multi-grained link you can use the whole vector as one interrelated 32 bit vector or split the whole vector into sub-channels. In this configuration each strip can be one single sub-channel or a group of strips can be gathered to a single sub-channel of de--sired bit-width. You just have - in respect of hardware costs - to consider that one sub-channel has to fit into one multi-grained link.

Multi -grained Routing

In order to route multi-grained channels it's necessary to use the coarse grained links to support the routing process. The idea is to route two links in parallel, one coarse-grained link to support multi -grained routing and one multi-grained link, which will contain the final multi-grained stream. Therefore we defined a two packet routing instruction with needed data fields . The first instruction packet contains - compared to coarse-grained routing instruction - additional bit mask to specify used multi - grained sub-links and multi-grained link ID to identify the associated multi -grained link. The other features like described above - optimum bit, speed path, priority routing - are support in this routing mode as well. The routing process within the RU is performed similar to the coarse-grained routing.
The first packet which arrives in a cell is analyzed by the InRegCtrl and a request is generated and forwarded to the InCtrl. InCtrl forwards the request to the RoutCtrl and wait for the acknowledgement. If RoutCtrl finds one possible routing direction, the InCtrl gets the successful acknowledgement and the temporal routing will be established by the RoutCtrl. *Next,* the actual job will be stored within the QueueRRP and the InCtrl waits for

the acknowledgement from the successor cell. If RoutCtrl is not able to find a possible routing, the InCtrl gets negative acknowledgement and which will be forwarded to the associated InRegCtrl, which generates the route unable signal to the predecessor cell and quits the routing process within this cell.
If the successor cell signals successful routing, the InRegCtrl clears the related entry in the QueueRRP and finishes the routing. If the successor cell is not able to establish a rout to the destination cell, it generates negative acknowledgement signal. Hereupon, the InCtIr starts new request to the RoutCtrl and handle the responses as described above.
The difference between the coarse-grained routing and multi-grained routing lies in the handling of the multi-grained interconnect matrix. Each strip of a multi-grained link is handled separately. The RoutCtrl forwards the switch request to the strip matcher. Strip matcher has the job to analyze the input strips and to match them to the output link according to already used strips. What strip matcher is doing is to map the problem of strip matching into the time domain and switches the needed switchboxes for each strip separately one after another.

Routing packet for coarse-grained streams:



	Value	Comments
	1	instruction-packet
31 30	00	ID: Routing-packet for coarse-grained streams
29 28	XX	Priority-level : higher value results in higher priority
	XX	Speed path : Reference value
25 24	XX	Speed path : Counter
23	X	Optimum bit (OptBit) : 1 enabled; 0 disabled
22 ... 19	XXXX	FM output address within the source cell
...	XXXX	FM input address within the destination cell
14	X	Use fine-grained links: 1 = yes, 0 = no
13 ... 8		Reserved
...	X...X	Destination cell coordinates: x-coordinate
3 ... 00	X..X	Destination cell coordinates : y-coordinate

Routing Instruction for multi-grained streams (first packet):

31
30
29
28
25
24
23
22
03
02
01
00

	Value	Comments
	1	Instructions -packet
31 30	01	ID: Routing-instruction multi-grained streams (first packet)
29 28	XX	Priority-level: higher value results in higher priority
	XX	Speed path: Reference value
25 24	XX	Speed path: Counter
23	X	Optimum bit (OptBit) : 1 enabled; 0 disabled
22	-	Reserved

21	..	[illegible]	XXX	ID of the input stream of the multi-grained link
[illegible]	..	[illegible]	XXX	8 bit strips mask: 1 = selected; 0 = not selected
[illegible]	..	[illegible]	X...X	1 bit strips mask- 1 = selected; 0 = not selected
[illegible]	..	[illegible]	XXXX	Destination cell coordinates : x-coordinate
3		0	XXXX	Destination cell coordinates : y-coordinate

Second packet of the routing instruction for multi-grained streams:

▨	31	30	29	28	▨	▨	13	12	11	10	09	08	07	06	05	04	00	▨

			Value	*Comments*
▨			1	Instructions-packet
31	...	30	10	ID: Routing-instruction multi-grained streams (first packet)
29	28		X	Reserved
27	...	[illegible]	XXX	Destination cell 8 bit strips mask: 1 = selected; 0 = not selected
24	...	17	X...X	Destination cell 1 bit strips mask: 1 = selected; 0 = not selected
▨	...	▨	XXX	Multi-grained FM input port address of the destination cell
13	...	11	XXX	Source cell 8 bit strips mask : 1 = selected; 0 = not selected
10	...	3	X...X	Source cell 1 bit strips mask : 1 = selected; 0 = not selected
2	..	0	XXX	Multi-grained FM output port address of the source cell

End packet instruction:

▨	31	30	29	28	27	26	25	24	23	▨	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

			Value	*Comment*
▨			1	Instruktions-Paket
31	...	30	11	ID: End packet for logical stream releasing
29			X	Coarse-/fine-grained releasing: 1 coarse-grained, 0 fine-grained
28			X	Local/global route release process: 1 = local, 0 = global
27	...	23	- ... -	Reserved
▨	...	▨	XXXX	PM output address within the source cell
18	..	14	- ... -	Reserved
13	...	11	XXX	Source cell 8 bit strips mask: 1 = selected; 0 = not selected
10	..	3	X .X	Source cell 1 bit strips mask: 1 = selected; 0 = not selected
2	..	0	XXX	Multi-grained FM output port address of the source cell

Data packet:

■	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00

	Value	Comments
■	0	Data packet
31 .. 0	X...X	Application data

Figures relating to improved way of routing : Pig . 37, Fig . 38, Fig . 39, Fig . 40, Fig . 41, Fig . 42, Fig. 43, Fig . 44, Fig . 45, Fig . 46.

Appendix t o PACT48/PCTE

Using Function Folding to Improve Silicon Efficiency of Reconfigurable Arithmetic Arrays

Authors: (Removed for blind review)

Abstract

This paper presents Function Folding, a design principle to improve the silicon efficiency of reconfigurable arithmetic (coarse-grain) arrays. Though highly parallel implementations of DSP algorithms have been demonstrated on these arrays, the overall silicon efficiency of current devices is limited by both the large numbers of ALUs required in the array and by the only moderate speeds which are achieved. The operating frequencies are mainly limited by the requirements of non-local routing connections. We present a novel approach to overcome these limitations; In Function Folding, a small number of distinct operators belonging to the same configuration are folded onto the same ALU, i. e. executed sequentially on one processing element. The ALU is controlled by a program repetitively executing the same instruction sequence. Data only required locally is stored in a local registerfile. This sequential approach uses the individual ALU resources more efficiently, while all processing elements of the array work in parallel as in current devices. Additionally, the ALUs and local registers can be clocked with a higher frequency than the (non-local) routing connections. Overall, a higher computational density than in current devices results.

1 Introduction

Field-Programmable Gate Arrays (FPGAs) are used as a flexible, programmable alternative to Application Specific Integrated Circuits (ASICs) for bit-oriented applications. They combine low MIE costs with fast time-to-market [I]. Similarly, reconfigurable arithmetic arrays - based on coarse-grain ALUs rather than bit-level lookup tables - are such an alternative for word-level, arithmetic applications. There are several research projects (e. g. Rapid [2], KressArray [1,3]) as well as commercial developments (e. g. PACT XPP Technologies [4], Morphotech [5], Elixent [6]) in this area. However, these architectures have not seen widespread use yet though highly parallel implementations of DSP algorithms have been demonstrated on them. One apparent reason for this is the limited silicon efficiency of current devices, resulting in both a large number of ALUs required in the array and in only moderate speeds being achieved. The operating frequencies are mainly limited by the requirements of non-local routing connections.

We present an extension of PACT XPP Techonologies [1]*eXtreme Processing Platform (XPP)* [4] which overcomes these limitations: Rather than executing a fixed operation on an ALU for the entire duration of a configuration, a small number of distinct operators belonging to the same configuration are folded onto the same ALU, i. e. executed sequentially on the same processing element (PE). The ALU is controlled by a program repetitively executing the same instruction sequence. Data only required locally is stored in a local register file. This sequential approach uses the individual ALU resources more efficiently, while all processing elements of the array work in parallel as in current devices. Since external data transfers are not required in every PE clock cycle, the ALUs and local registers can be clocked with a higher frequency than the (non-local) routing connections. This *ALU overclocking* technique is also justified by the continuous trend to higher integration densities: New technology generations provide smaller and smaller transistors, but the wires have higher relative capacities which make the busses slower and more power-consuming.

Despite these significant architectural changes, existing XPP programs can be automatically mapped to this extended architecture. Overall, a higher computational density than in current devices results.

The remainder of this paper is organized as follows:
First, we describe the current PACT XPP architecture. Next, Section 3 describes the functionality and hardware design of the new Function Folding PE, Section 4 elaborates the application mapping methods, and Section 5 presents preliminary results. Finally, our approach is compared to related work, conclusions are drawn, and future work is outlined.

2 XPP Architecture Overview

The current XPP architecture [4] is based on a 2-D array of coarse-grain, adaptive processing elements (PEs), internal memories, and interconnection resources. A 24-bit prototype chip with 64 ALUs and 16 internal memories was built by PACT XPP Technologies. A development board for this XPP64A chip is available.

PACT also provides a complete development tool suite consisting of a placer and router, a simulator, and a visualizer. The tools use the proprietary *Native Mapping Language* (NML), a structural language with reconfiguration primitives. A C frontend is provided as well.

2.1 Array Structure

Fig. 47 shows the basic structure of a simple XPP core. For demonstration purposes, it contains only 9 PEs and 6 internal RAMs. The core comprises a 3 x 3 square of PEs in the center and one column of independent internal memories on each side. There are two *I/O* units which can either be configured as ports for streaming data or as interfaces for external RAM access. The core of a PE is an ALU which performs common arithmetic and logical operations, comparisons, and special operations such as counters. In each configuration, each PE performs one dedicated operation. Each line in the figure represents a set of segmented busses which can be configured to connect the output of a PE with other PEs' inputs. The array is attached to a *Configuration Manager* (CM) responsible for the runtime management of configurations, i. e. for downloading configuration data from external memory into the configurable resources of the array. Besides a finite state machine, the CM has cache memory for storing or pre-fetching configuration data.

2.2 Data and Event Synchronization

The interconnection resources consist of two independent sets of busses: data busses (with a device specific bit-width) and one-bit wide event busses. The XPP busses are not just wires to connect logic: a ready / acknowledge protocol implemented in hardware synchronizes the data and events processed by the PEs. Hence a PE operation is performed as soon as all necessary input values are available and the previous result has been consumed. Thus it is possible to map a dataflow graph directly to the array, and to pipeline input data streams through it. No data is lost even during pipeline stalls. Special dataflow operations for stream merging, multiplexing etc. are provided as well.

2.3 Configuration

Compared to FPGAs, XPP cores can be configured rapidly due to their coarse-grain nature: Only opcodes and connections have to be set. Furthermore, only those array objects actually used need to be configured.

The configuration time can be reduced by prefetching mechanisms: during the loading of a configuration onto the array another configuration can be loaded to the CM cache. Thus it must not be loaded from external memory when it is requested. The same is true if a configuration has been used before and its configuration data is still in the CM cache.

3 Function Folding PE

We now describe the functionality and hardware design of an extended XPP PE, the *Function Folding Processing Element.*

3.1 Function Folding Example

Let us first consider a simple example: An address *adr* is computed from a constant offset *offs* and coordinates *x* and *y* as follows: $adr = offs + x + 256 * y$. In an XPP implementation based on simple PEs as shown in Fig. 47, this computation is normally directly mapped to the dataflow graph in Fig. 49a. Each adder and multiplier is mapped to its own ALU. Therefore a new address can be computed every cycle. However, as mentioned in Section 1, the operating frequency is limited by the bus connections, not by the ALU itself.

For a higher silicon efficiency, i. e. for more operations per square millimeter and second, the ALUs have to be clocked faster. This could be achieved by more pipeline registers in the busses. But they unfortunately increase the chip area and power consumption and reduce the throughput if the dataflow graph contains cycles. In our approach, we rather operate the busses at a moderate frequency (as in the current XPP cores) and increase the ALU's clock rate locally inside a PE. This *n-fold overclocking* allows to schedule *n* ALU operations in one bus cycle (for a small number *n).* We call these groups of operations *clusters.* The significant reduction in the number of PEs required justifies the hardware overhead incurred. While sticking to the successfull paradigm of reconfigurable "computing in space", this locally sequential approach optimizes the usage of the ALU resources.

By allowing different overclocking factors in the same device (e. g. $n = 2$ and $n = 4$), different local time-space tradeoffs are possible. For $n = 4$, in our example, all operations in the dataflow graph can be clustered, i. e. executed on the same ALU, even if the multiplication requires two cycles. For $n = 1$, only the two adders can be clustered. This results in twice the area, but also doubles the throughput compared to $n = 4$.

Apart from a program controller executing the *n* instructions repetitively, a small internal register file to feed intermediate results back to the ALU inputs is required in the PE. This local feedback loop allows to implement dataflow graphs with cycles containing up to *n* operators without reducing the overall throughput.

3.2 Hardware Design

The hardware design sketched in Fig. 48 performs Function Folding as described above. As the simple PEs in Fig. 47, the new PE communicates with the interconnect network via data and event input and output ports which follow the ready/acknowledge protocol. The ports also synchronize the fast internal PE clock with the ra-times slower bus clock. Input data is stable during the entire bus clock cycle, i. e. can be sampled in any of the internal PE clock cylces. And output data is copied to the bus registers at the beginning of a bus cycle. A Function Folding PE requires more ports than a simple PE since it executes an entire cluster of operations. But it does not require «-times more ports than the simple PE since the number of external connections is often quite limited due to constant inputs (which can be loaded to internal registers at configuration time) and local connections within a cluster. A good clustering algorithm minimizes the number of external connections. As illustrated by the dotted box around the operators in Fig. 49b, only two input ports and one output port are required for the example cluster for $n = 4$.

In detail, the PE in Fig. 48 works as follows: A small program counter (PC) repeatedly iterates through the configured instructions in the instruction store. In each PE cycle it selects the ALU opcode and controls the multiplexors selecting the ALU inputs. Either an input port or an entry of the internal register file can be used. The ALU outputs can be written back to the internal register file or to an output port or to both. The entire design is kept as simple and small as possible to just support function folding. No other control structures are possible. Both the number of input and output ports and the number of internal registers will be about n. Therefore we can choose a very fast implementation of the register file just using registers and multiplexors. Given the small number of ports and registers, the entire fetch/execute/store process can be performed in one cycle. The only exception is the multiplier operation which takes two cycles. The controller (FSM) stalls the program execution if an external input is not available or if an external output port is full due to a downstream pipeline stall. Note that event ports and registers are omitted in Fig. 48 for clarity. Events can be used and stored internally and externally like data.

Returning to Fig. 49b, we can now present the simple PE program for the address generation cluster. The mapping of connections to ports and registers is indicated in the figure. We assume that registers r1 and r2 have been initialized with the constant values *offs* and 256, respectively, at configuration time. The following assembler code, executed repetitively, describes the cluster:

```
add r3 <- r1, i1
mul r4 <- r2, i2
add o1 <- r3, r4
```

4 Application Mapping

Fig. 50 shows the tool flow of the extended XPP architecture. It is very similar to the current tool flow implemented in the xnriap program [4]. Only the phases represented by the shaded boxes are added. The following phases already exist in the current XPP tool flow:

- *Cfrontend (optional):* Generates structural NML code (cf. Section 2) from a subset of standard C.
- *NMLparser:* Parses the input NML file and maps it to XPP operators.
- *Place and Route:* Places the PEs (i. e. operators in the current architecture) on the XPP array and routes the connections.
- *Binary Generation:* Generates an XBIN binary file.

For Function Folding, an additional *Operator Clustering* phase is required which defines the operators mapped to one PE. Though the clusters could be defined manually by annotations in the NML file, an automatie clustering algorithm is required to simplify programming, to use the C frontend, and to map existing NML code. It is described in the next section. Furthermore, PE program code needs to be generated for each cluster as described in Section 4.2. Obviously the Place and Route and Binary Generation phases have to be adapted, too.

4.1 Operator Clustering

The operator clustering problem for Function Folding PEs is similar to the graph covering problems encountered in code generators for conventional processors and module mapping for FPGAs, e. g. [7]. Therefore we investigated these algorithms first. The efficient dynamic-programming algorithm used in [7] and similar approaches is actually a tree-covering algorithm. It generates optimal coverings for operator trees. But it cannot handle arbitrary dataflow graphs. Hence a preprocessing phase which removes feedback cycles and fanout edges from the original graph is required. The result is a forest of trees which can be covered efficiently. However, the optimal tree covering results are not optimal for the original dataflow graph .

Now consider the operator clustering problem at hand: We need to find a solution with the minimal number of clusters which conforms to the restrictions of the Function Folding PEs, i. e. the over-clocking factor *n*, and the number of ports and internal registers. Additionally, cycles should be processed within a cluster whenever possible (to avoid reduced throughput caused by external routing delays), and the number of external connections should be minimized. Unfortunately these quality criteria are not visible in the output of the tree covering preprocessing phase, i. e. after the removal of cycles and fanout edges. Hence we do not apply tree covering for operator clustering.

Instead, we developed an algorithm operating on the original graph. To reduce the complexity, we only consider clustering operators which are connected since only these clusters use internal registers and reduce the number of external ports. In an additional postprocessing phase, unconnected clusters can be merged later if they are placed next to each other.

In the first algorithm phase, all connected clusters are explicitly generated. Note that the number of possible unconnected clusters would be exponential in the number of operators.

In the second phase, the optimal combination of clusters covering the entire input graph has to be determined. Unfortunately the number of all possible combinations of clusters is exponential. Hence it cannot be searched exhaustively. Instead, the main loop of the algorithm operates on an increasing subset of the operators, generating and storing an optimal clustering of the subset, until the optimal clustering of the entire operator graph has been computed. The algorithm exploits the fact that partial optimal solutions are contained in the complete optimal solution. In this way we do not need to compute optimal clusterings for *all* subsets. Because the optimal clustering of a new subset depends on other subsets which might not have been computed before, some recursive calls which may lead to an exponential runtime are required. However, we found that the runtime is in the range of a few minutes for an overclocking factor $n <= 4$ and for an operator number $k < 50$. For larger problem sizes we tested the following heuristics:

- Remove large feedback cycles (with more than *n* operators) from the graph. Then small cycles are still executed within a cluster and only a few possible clusters are exlcuded, but the number of recursive calls is largely reduced.
- Do not compute the *best* solution in recursive calls, but only *the first* clustering of the subset which is computed. By applying larger clusters before smaller ones, the algorithm computes a nearly optimal solution anyway.

With these extensions we could quickly cluster realistic dataflow graphs with up to 150 operators. For the cases we tested, the heuristics produced clusterings which were very near or equal to the optimum (i. e. they had only a few more clusters).

Note that we restricted the number of operations in a PE program to *n* in the previous discussions. This is reasonable since we normally do not want to extend the PE program execution over more than one bus cycle. However, if a PE can be programmed to execute more than *n* operations, those operations which are not throughput-critical can be combined in larger clusters. This further reduces the required number of PEs without impacting the overall throughput.

4.2 PE Code Generation

After operator clustering, the PE program code for every cluster is generated, cf. the assembler code in the example in Section 3. A simplified version of conventional register allocation is used to map internal connections to internal registers. The instructions can be directly extracted from the dataflow graph of the cluster.

5 Results

5.1 PE Speed and Area

The area of a Function Folding PE is estimated to be about 15 % to 25 % larger than the area of the corresponding simple PE, depending on the number of ports and registers. For 16-bit datapaths, preliminary synthesis results achieve a PE frequency of 400 - 500 MHz for a 130 run silicon process.

Note to the reviewer: For thefinal version of thispaper, we expect topresent detailed analysis results from a PE hardware implementation currently under way.

5.2 Complex FIR Application Analysis

This section demonstrates the implementation of a typical DSP algorithm, a FIR filter operating on complex numbers, on Function Folding PEs. Consider one FIR filter cell which computes the output $Z = X * C + Y$ from a constant C and inputs X and Y. All values are complex, i. e. *(Zre,Zim) = (Xre * Cre-Xim*Cim+Yre, Xre*Cim+Xim*Cre+Yim).* Fig. 51a shows the corresponding dataflow graph. It contains eight operators which can be folded to three clusters for an overclocking factor of $n = 4$, as indicated by the dotted boxes. Fig. 51b shows the resulting cluster dataflow graph. All clusters fully utilize the PEs, i. e. use all four PE clock cycle.

Let us now compare the silicon efficiency of an implementation on a current XPP device (FIRcurr) with one based on Function Folding PEs (FIRnew), As outlined above, we estimate $F_{PE} = 400$ MHz and $F_{bus} = 100$ MHz for $n = 4$. F_{bus} is also the operating frequency of the current architecture. Filters built from the given FIR cells can easily be fully pipelined for both implementations, as can be seen from Fig. 51a and b. Hence both implementations have the same performance: They generate outputs at a rate of 100 MHz.

The area of a Function Folding PE is estimated as $A_{FFPE} = 1.2 \times A_{currPE}$, i. e. 20 % larger than current PEs. The area ratio for the two filter implementations is as follows:

$$\frac{A_{FIRnew}}{A_{FIRcurr}} = \frac{3 \times A_{FFPE}}{8 \times A_{currPE}} = \frac{3 \times 1.2 \times A_{currPE}}{8 \times A_{currPE}} = 0.45$$

This rough estimation shows that the new implementation is more than twice as area-efficient than the old one without requiring more pipelining registers in the external busses. The overall silicon efficiency is more than doubled.

5.3 Benchmark Mapping Results

In order to determine the general applicability of Function Folding, the algorithm described in Section 4.1 was applied to a benchmark of 43 legacy XPP configurations from a wide range of application areas. We determined the average *cluster utilization,* i. e. the number of PE cycles being used by the repetitive PE program. [1] This value is a good indication of the effectiveness of Function Folding.

Table 1 shows the results for $n = 2$ and $n = 4$ with varying port numbers. The number of internal registers was not yet restricted for this evaluation. The results for four input and output data and event ports (1.78 for $n = 2$ and 3.05 for $n = 4$) show that the Function Folding PE resources can be exploited efficiently for average XPP configurations. The table shows that using six data ports increases the cluster utilization only insignificantly. On the other hand, using fewer data ports distinctly decrea-

[1] Note that a high cluster utilization does not guarantee that the PE program can be executed every bus cycle. The overall PE utilization in an application also depends on the availability of input data and on the overall throughput of all PBs.

ses the utilization. We will combine hardware implementation results detailing the area requirements of the ports with the cluster utilization numbers to determine the PE parameters which yield the best overall silicon efficiency.

n	DI	DO	EI	EO	CU
2	6	6	4	4	1.79
2	4	4	4	4	1.78
2	2	2	4	4	1.57
2	4	4	2	2	1.75
2	2	2	2	2	1.53
4	6	6	4	4	3.06
4	4	4	4	4	3.05
4	2	2	4	4	2.25
4	4	4	2	2	2.80
4	2	2	2	2	2.12

Table 1: Average cluster utilization (CU) in XPP benchmark, *n:* overclocldng factor; DMDO: number of data input/output ports; EI/EO: number of event input/output ports.

6 Related Work

Though there are several projects onreconfigurable arithmetic arrays as mentioned in Section 1, to our knowledge there are no solutions similar to Function Folding in the literature. The following architectures differ considerably from our approach, but also allow to quickly change the operations performed by a PE.

The RAW microprocessor [8] also contains a cluster of processing elements, but they are rather complex processors. Therefore their programs cannot be generated automatically as easily as the Function Folding PE programs. The RAW architecture resembles more amultiprocessor on a chip.

On the other hand, the MorphoSys architecture [5] follows a SIMD approach. All PEs in a row or column are controlled by a (global) program and execute the same instruction. This makes the PEs simpler, but the SIMD principle considerably restricts the available computations which can be executed. The array is also much harder to program.

Finally, *multi-context devices* provide two or more complete configuration contexts [9]. This technique is adapted from multi-context FPGAs. However, it does not allow frequent reconfigurations since the shadow configurations first have to be loaded completely. The configurations are completely independent. Multicontext devices hide the configuration latency to a certain extent, but do not overcome the general efficiency problems of coarse-grain reconfigurable architectures.

7 Conclusions and Future Work

We have presented the architecture and functionality of the Function Folding Processing Element for an enhanced PACT XPP architecture. Preliminary analyses of both a hardware implementation and applications mapped to this architecture indicate that Function Folding significantly increases the silicon efficiency compared to current reconfigurable arithmetic arrays and has the potential to reduce the power consumption.

After the implementation of a Function Folding PE and the analysis of its parameters, future work will include the evaluation of paths which are not throughput-critical as mentioned at the end of Section 4.1. Integrated clustering and place-and-route algorithms will be explored. We also consider developing a direct compiler from C to Function Folding PEs which might exploit their capabilities better then the current design flow via NML.

References

[1] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proc. Design, Automation and Test in Europe, 2001.*

[2] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. Architecture design of reconfigurable pipelined datapaths. In *Proc. 20th Anniversary Conference on Advanced Research in VLSI,* Atlanta, GA, March 1999.

[3] R. Hartenstein, R. Kress, and H. Reinig. A new FPGA architecture for word-oriented datapaths. In *Proc. Field-Programmable Logic; 4th International Workshop.* Springer-Verlag, September 1994.

[4] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP - a self-reconfigurable data processing architecture. *The Journal of Supercomputing,* 26(2), September 2003.

[5] M.-H. Lee, H. Singh, G. Lu, N. Bagherzadeh, and F. J. Kurdahi. Design and implementation of the MorphoSys reconfigurable computing processor. *Journal of VLSI and Signal Processing-Systems for Signal, Image and Video Technology,* March 2000. March 2000.

[6] T. Stansfield. Using multiplexers for control and data in D-Fabrix. In *Field Programmable Logic and Applications,* LNCS 2778, pages 416-425. Springer, 2003.

[7] T. J. Callahan, P. Chong, A. DeHon, and J. Wawrzynek. Fast module mapping and placement for datapaths in FPGAs. In *Proc. FPGA '98,* Monterrey, CA, 1998.

[8] M. B. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro,* March/April 2002.

[9] B. Salefski and L. Caglar. Re-configurable computing in wireless. In *Proc. 38th Design Automation Conference,* Las Vegas, NV, June 2001.

Appendix to PACT 48/PCT E

Using Function Folding to Improve Silicon Efficiency of Reconfigurable Arithmetic Arrays

Authors: (Removed for blind review)

Abstract

27MS paper presents Function Folding, a design principle to improve the silicon efficiency of reconfigurable arithmetic (coarse-grain) arrays. Though highly parallel implementations of DSP algorithms have been demonstrated on these arrays, the overall silicon efficiency of current devices is limited by both the large numbers of AL Us required in the array and by the only moderate speeds which are achieved. The operating frequencies are mainly limited by the requirements of non-local routing connections. We present a novel approach to overcome these limitations: In Function Folding, a small number of distinct operators belonging to the same configuration are folded onto the same ALU, i. e. executed sequentially on one processing element. The ALU is controlled by a program repetitively executing the same instruction sequence. Data only required locally is stored in a local register file. This sequential approach uses the individual AL U resources more efficiently, while all processing elements of the array work in parallel as in current devices. Additionally, the AL Us and local registers can be clocked with a higher frequency than the (non-local) routing connections. Overall, a higher computational density than in current devices results.

1 Introduction

Field-Programmable Gate Arrays (FPGAs) are used as a flexible, programmable alternative to Application Specific Integrated Circuits (ASICs) for bit-oriented applications. They combine low NRE costs with fast time-to-market [I]. Similarly, reconfigurable arithmetic arrays - based on coarse-grain ALUs rather than bit-level lookup tables - are such an alternative for word-level, arithmetic applications. There are several research projects (e. g. Rapid [2], KressArray [1, 3]) as well as commercial developments (e. g. PACT XPP Technologies [4], Morphotech [5], Elixent [6]) in this area. However, these architectures have not seen widespread use yet though highly parallel implementations of DSP algorithms have been demonstrated on them. One apparent reason for this is the limited silicon efficiency of current devices, resulting in both a large number of ALUs required in the array and in only moderate speeds being achieved. The operating frequencies are mainly limited by the requirements of non-local routing connections.

We present an extension of PACT XPP Techonologies' *eXtreme Processing Platform (XPP)* [4] which overcomes these limitations: Rather than executing a fixed operation on an ALU for the entire duration of a configuration, a small number of distinct operators belonging to the same configuration are folded onto the same ALU, i. e. executed sequentially on the same processing element (PE). The ALU is controlled by a program repetitively executing the same instruction sequence. Data only required locally is stored in a local register file. This sequential approach uses the individual ALU resources more efficiently, while all processing elements of the array work in parallel as in current devices. Since external data transfers are not required in every PE clock cycle, the ALUs and local registers can be clocked with a higher frequency than the (non-local) routing connections. This *ALU overclocking* technique is also justified by the continuous trend to higher integration densities: New technology generations provide smaller and smaller transistors, but the wires have higher relative capacities which make the busses slower and more power-consuming.

Despite these significant architectural changes, existing XPP programs can be automatically mapped to this extended architecture. Overall, a higher computational density than in current devices results.

The remainder of this paper is organized as follows: First, we describe the current PACT XPP architecture. Next, Section 3 describes the functionality and hardware design of the new Function Folding PE, Section 4 elaborates the application mapping methods, and Section 5 presents preliminary results. Finally, our approach is compared to related work, conclusions are drawn, and future work is outlined.

2 XPP Architecture Overview

The current XPP architecture [4] is based *on* a 2-D array of coarse-grain, adaptive processing elements (PEs), internal memories, and interconnection resources. A 24-bit prototype chip with 64 ALUs and 16 internal memories was built by PACT XPP Technologies. A development board for this XPP64A chip is available.

PACT also provides a complete development tool suite consisting of a placer and router, a simulator, and a visualizer. The tools use the proprietary *Native Mapping Language* (NML), a structural language with reconfiguration primitives. A C frontend is provided as well.

2.1 Array Structure

Fig. 1 shows the basic structure of a simple XPP core. For demonstration purposes, it contains only 9 PEs and 6 internal RAMs. The core comprises a 3 x 3 square of PEs in the center and one column of independent internal memories on each side. There are two I/O units which can either be configured as ports for streaming data or as interfaces for external RAM access. The core of a PE is an ALU which performs common arithmetic and logical operations, comparisons, and special operations such as counters. In each configuration, each PE performs one dedicated operation. Each line in the figure represents a set of segmented busses which can be configured to connect the output of a PE with other PEs' inputs. The array is attached to a *Configuration Manager* (CM) responsible for the runtime management of configurations, i. e. for downloading configuration data from external memory into the configurable resources of the array. Besides a finite state machine, the CM has cache memory for storing or pre-fetching configuration data.

2.2 Data and Event Synchronization

The interconnection resources consist of two independent sets of busses: data busses (with a device specific bit-width) and one-bit wide event busses. The XPP busses are not just wires to connect logic: a ready /acknowledge protocol implemented in hardware

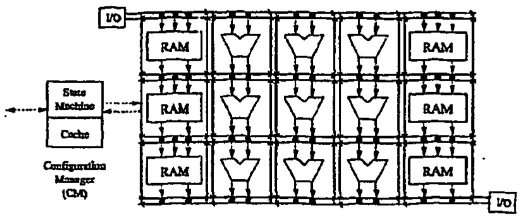


Figure 1: Simplified structure of an XPP array.

synchronizes the data and events processed by the PEs. Hence a PE operation is performed as soon as all necessary input values are available and the previous result has been consumed. Thus it is possible to map a dataflow graph directly to the array, and to pipeline input data streams through it. No data is lost even during pipeline stalls. Special dataflow operations for stream merging, multiplexing etc. are provided as well.

2.3 Configuration

Compared to FPGAs, XPP cores can be configured rapidly due to their coarse-grain nature: Only opcodes and connections have to be set. Furthermore, only those array objects actually used need to be configured.

The configuration time can be reduced by prefetching mechanisms: during the loading of a configuration onto the array another configuration can be loaded to the CM cache. Thus it must not be loaded from external memory when it is requested. The same is true if a configuration has been used before and its configuration data is still in the CM cache.

3 Function Folding PE

We now describe the functionality and harware design of an extended XPP PE, the *Function Folding Processing Element*

3.1 Function Folding Example

Let us first consider a simple example: An address *adr* is computed from a constant offset *offs* and coordinates x and y as follows: $adr = offa+5+256*y$. In an XPP implementation based on simple PEs as shown in Fig. 1, this computation is normally directly mapped to the dataflow graph in Fig. 3(a). Each adder and multiplier is mapped to its own ALU. Therefore a new address can be computed every cycle. However,

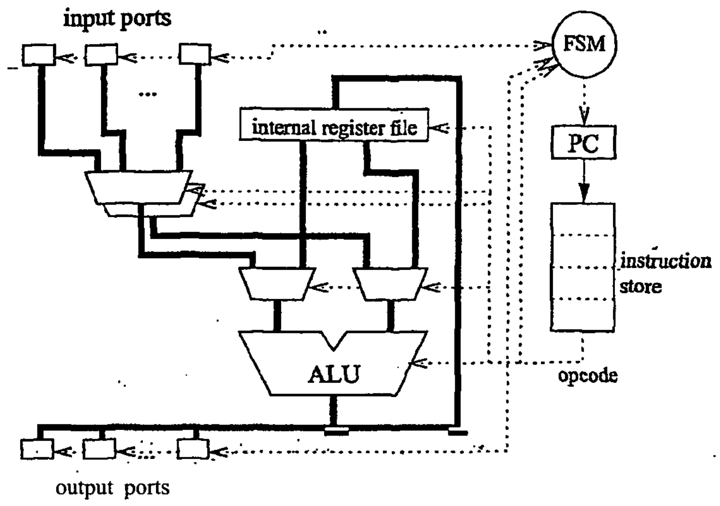


Figure 2: Function Folding Processing Element.

as mentioned in Section 1, the operating frequency is limited by the bus connections, not by the ALU itself.

For a higher silicon efficiency, i. e. for more operations per square millimeter and second, the ALUs have to be clocked faster. This could be achieved by more pipeline registers in the busses. But they unfortunately increase the chip area and power consumption and reduce the throughput if the dataflow graph contains cycles. In our approach, we rather operate the busses at a moderate frequency (as in the current XPP cores) and increase the ALU's clock rate locally inside a PE. This *n-fold overclocking* allows to schedule *n* ALU operations in one bus cycle (for a small number *n*). We call these groups of operations *dusters.* The significant reduction in the number of PEs required justifies the hardware overhead incurred. While sticking to the successfull paradigm of reconfigurable "computing in space", this locally sequential approach optimizes the usage of the ALU resources.

By allowing different overclocking factors in the same device (e.g. $n = 2$ and $n = 4$), different local time-space tradeoffs are possible. For n = 4, in our example, all operations in the dataflow graph can be clustered, i. e. executed on the same ALU, even if the multiplication requires two cycles. For $n = 2$, only the two adders can be clustered. This results in twice the area, but also doubles the throughput compared to $n = 4$.

Apart from a program controller executing the *n* instructions repetitively, a small internal register file to feed intermediate results back to the ALU inputs is required in the PE. This local feedback loop allows to implement dataflow graphs with cycles containing up to *n* operators without reducing the overall throughput.

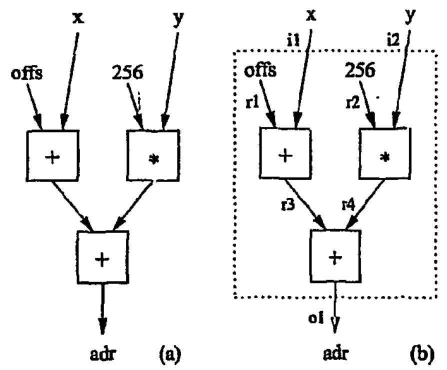


Figure 3: Address generation dataflow graph.

3.2 Hardware Design

The hardware design sketched in Fig. 2 performs Function Folding as described above. As the simple PEs in Fig. 1, the new PE communicates with the interconnect network via data and event input and output ports which follow the ready/acknowledge protocol. The ports also synchronize the fast internal PE clock with the ra-times slower bus clock. Input data is stable during the entire bus clock cycle, i.e. can be sampled in any of the internal PE clock cylces. And output data is copied to the bus registers at the beginning of a bus cycle. A Function Folding PE requires more ports than a simple PE since it executes an entire cluster of operations. But it does not require n-times more ports than the simple PE since the number of external connections is often quite limited due to constant inputs (which can be loaded to internal registers at configuration time) and local connections within a cluster. A good clustering algorithm minimizes the number of external connections. As illustrated by the dotted box around the operators in Fig. 3(b), only two input ports and one output port are required for the example cluster for $n = 4$.

In detail, the PE in Fig. 2 works as follows: A small program counter (PC) repeatedly iterates through the configured instructions in the instruction store. In each PE cycle it selects the ALTJ opcode and controls the multiplexors selecting the ALU inputs. Either an input port or an entry of the internal register file can be used. The ALU outputs can be written back to the internal register file or to an output port or to both. The entire design is kept as simple and small as possible to just support function folding. No other control structures are possible. Both the number of input and output ports and the number of internal registers will be about n. Therefore we can choose a very fast implementation of the register file just using registers and multiplexors. Given the small number of ports and registers, the entire fetch/execute/store process can be performed in one cycle. The only exception is the multiplier operation which takes two cycles. The controller (FSM) stalls the program execution if an external input is not available or if an external output port is full due to a downstream pipeline stall. Note that event ports and registers are omitted in Fig. 2 for clarity. Events can be used and stored internally and externally like data.

Returning to Fig. 3(b), we can now present the simple PE program for the address generation cluster. The mapping of connections to ports and registers is indicated in the figure. We assume that registers r1 and r2 have been initialized with the constant values *offs* and 256, respectively, at configuration time. The following assembler code, executed repetitively, describes the cluster:

```
add r3 <- ri,  i1
mul r4 <- x2,  12
add ol <- r3,  r4
```

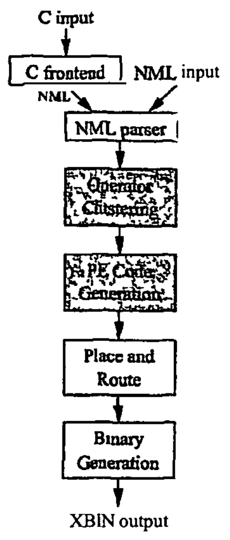


Figure 4: Extended XPP tool flow.

4 Application Mapping

Fig. 4 shows the tool flow of the extended XPP architecture. It is very similar to the current tool flow implemented in the xmap program [4]. Only the phases represented by the shaded boxes are added. The following phases already exist in the current XPP tool flow:

- *Cfrontend (optional):* Generates structural NML code (cf. Section 2) from a subset of standard C.
- JVMZi *parser:* Parses the input NML file and maps it to XPP operators.
- *Place and Route:* Places the PEs (Le. operators in the current architecture) on the XPP array and routes the connections.
- *Biηary Generation:* Generates an XBIN binary file.

For Function Folding, an additional *Operator Clustering* phase is required which defines the operators mapped to one PE. Though the clusters could be defined manually by annotations in the NML file, an automatic clustering algorithm is required to simplify programming, to use the C frontend, and to map existing NML code. It is described in the next section. Furthermore, PE program code needs to be generated for each cluster as described in Section 4.2. Obviously the Place and Route and Binary Generation phases have to be adapted, too.

4.1 Operator Clustering

The operator clustering problem for Function Folding PEs is similar to the graph covering problems encountered in code generators for conventional processors and module mapping for FPGAs, e.g. [7]. Therefore we investigated these algorithms first. The efficient dynamic-programming algorithm used in [7] and similar approaches is actually a tree-covering algorithm. It generates optimal coverings for operator trees. But it cannot handle arbitrary dataflow graphs. Hence a preprocessing phase which removes feedback cycles and fanout edges from the original graph is required. The result is a forest of trees which can be covered efficiently. However, the optimal tree covering results are not optimal for the original dataflow graph.

Now consider the operator clustering problem at hand: We need to find a solution with the minimal number of clusters which conforms to the restrictions of the Function Folding PEs, i.e. the overclocking factor n, and the number of ports and internal registers. Additionally, cycles should be processed within a cluster whenever possible (to avoid reduced throughput caused by external routing delays), and the number of external connections should be minimized. Unfortunately these quality criteria are not visible in the output of the tree covering preprocessing phase, i.e. after the removal of cycles and fanout edges. Hence we do not apply tree covering for operator clustering.

Instead, we developed an algorithm operating on the original graph. To reduce the complexity, we only consider clustering operators which are connected since only these clusters use internal registers and reduce the number of external ports. In an additional postprocessing phase, unconnected clusters can be merged later if they are placed next to each other.

In the first algorithm phase, all connected clusters are explicitly generated. Note that the number of possible^unconnected clusters would be exponential in the number of operators.

In the second phase, the optimal combination of clusters covering the entire input graph has to be determined. Unfortunately the number of all possible combinations of clusters is exponential. Hence it cannot be searched exhaustively. Instead, the main loop of the algorithm operates on an increasing subset of the operators, generating and storing an optimal clustering of the subset, until the optimal clustering of the entire operator graph has been computed. The algorithm exploits the fact that partial optimal solutions are contained in the complete optimal solution. In this way we do not need to compute optimal clusterings for *all* subsets. Because the optimal clustering of a new subset depends on other subsets which might not have been computed before, some recursive calls which may lead to an exponential runtime are required. However, we found that the runtime is in the range of a few minutes for an overclocking factor $n <= 4$ and for an operator number $k < 50$. tbr larger problem sizes we tested the following heuristics:

- Remove large feedback cycles (with more than n operators) from the graph. Then small cycles are still executed within a cluster and only a few possible clusters are exlcuded, but the number of recursive calls is largely reduced.

- Do not compute the *best* solution in recursive calls, but only the *first* clustering of the subset which is computed. By applying larger clusters before smaller ones, the algorithm computes a nearly optimal solution anyway.

With these extensions we could quickly cluster realistic dataflow graphs with up to 150 operators. For the cases we tested, the heuristics produced clusterings which were very near or equal to the optimum (i.e. they had only a few more clusters).

Note that we restricted the number of operations in a PE program to n in the previous discussions. This is reasonable since we normally do not want to extend the PE program execution over more than one bus cycle. However, if a PE can be programmed to execute more than n operations, those operations which are not throughput-critical can be combined in larger clusters. This further reduces the required number of PEs without impacting the overall throughput.

4.2 PE Code Generation

After operator clustering, the PE program code for every cluster is generated, cf.', the assembler code in the example in Section 3. A simplified version of conventional register allocation is used to map internal connections to internal registers. The instructions can be directly extracted from the dataflow graph of the cluster.

5 Results

5.1 PE Speed and Area

The area of a Function Folding PE is estimated to be about 15 % to 25 % larger than the area of the corresponding simple PE, depending on the number of ports and registers. For 16-bit datapaths, preliminary synthesis results achieve a PE frequency of 400 - 500 MHz for a 130 nm silicon process.

Note to the reviewer: For the final version of this paper, we expect to present detailed analysis results from a PE hardware implementation currently under way.

5.2 Complex FIR Application Analysis

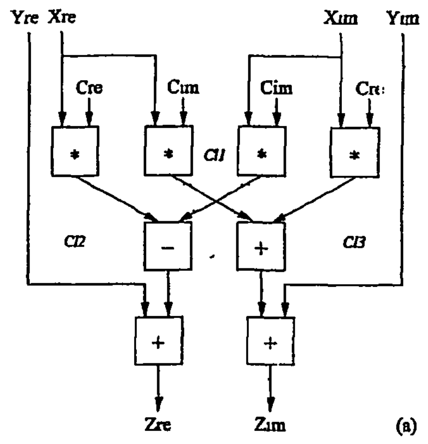
This section demonstrates the implementation of a typical DSP algorithm, a FIR filter operating on complex numbers, on Function Folding PEs. Consider one FIR filter cell which computes the output $Z = X * C + Y$ from a constant C and inputs X and Y. All values are complex, i.e. *(Zre, Zim) = (Xre * Cre—Xim*Cim+Yre,Xre*Cim+Xim*Cre+Yim)*. Fig. 5(a) shows the corresponding dataflow graph. It contains eight operators-which can be folded to three clusters for an overclockmg factor of $n = 4$, as indicated by the dotted boxes. Fig. 5(b) shows the resulting cluster dataflow graph. All clusters fully utilize the PEs, i.e. use all four PE clock cycle.

Let us now compare the silicon efficiency of an implementation on a current XPP device (FIRcurr) with one based on Function Folding PEs (FIRnew). As outlined above, we estimate $FPE = 400$ MHz and $Fbus = 100$ MHz for $n = 4$. F_{bus} is also the operating frequency of the current architecture. Filters built from the given FIR cells can easily be fully pipelined for both implementations, as can be seen from Fig. 5(a) and (b). Hence both implementations have the same performance: They generate outputs at a rate of 100 MHz.

The area of a Function Folding PE is estimated as $AFFPE = 1.2 X A_{currPE}$, i.e. 20 % larger than current PEs. The area ratio for the two filter implementations is as follows:

$$\frac{A_{FIRnew}}{A_{FIRcurr}} = \frac{3 \text{ X } A_{FFPE}}{8 \text{ x } A_{currPE}} = \frac{3 \text{ X } 1.2 \text{ X } A_{currPE}}{8 \text{ X } A_{currPE}} = 0.45$$

This rough estimation shows that the new implementation is more than twice as area-efficient than the old one without requiring more pipelining registers in the external busses. The overall silicon efficiency is more than doubled.



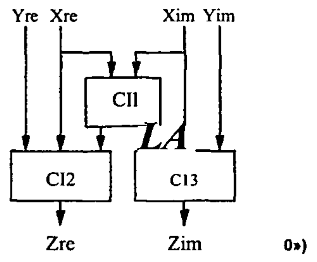


Figure 5: Complex FIR filter cell.

5.3 Benchmark Mapping Results

In order to determine the general applicability of Function Folding, the algorithm described in Section 4.1 was applied to a benchmark of 43 legacy XPP configurations from a wide range of application areas. We determined the average *cluster utilization,* i.e. the number of PE cycles being used by the repetitive PE program. [1] This value is a good indication of the effectiveness of Function Folding.

Table 1 shows the results for n = 2 and $n = 4$ with varying port numbers. The number of internal registers was not yet restricted for this evaluation. The results for four input and output data and event ports (1.78 for $n = 2$ and 3.05 for $n = 4$) show that the Function Folding PE resources can be exploited efficiently for average XPP configurations. The table shows that using six data ports increases the cluster utilization only insignificantly. On the other hand, using fewer

[1]Note that a high cluster utilization does not guarantee that the PE program can be executed every bus cycle. The overall PE utilization in an application also depends on the availability of input data and on the overall throughput of all PEs

n	DI	DO	EI	EO	CU
2	6	6	4	4	1.79
2	4	4	4	4	1.78
2	2	2	4	4	1.57
2	4	4	2	2	1.75
2	2	2	2	2	1.53
4	6	6	4	4	3.06
4	4	4	4	4	3.05
4	2	2	4	4	2.25
4	4	4	2	2	2.80
4	2	2	2	2	2.12

Table 1: Average cluster utilization (CU) in XPP benchmark, *n:* overclocking factor; DI/DO: number of data input /output ports; EI/EO: number of event input/output ports.

data ports distinctly decreases the utilization. We will combine hardware implementation results detail- ing the area requirements of the ports with the cluster utiüzation numbers to determine the PE parameters which yield the best overall silicon efficiency.

6 Related Work

Though there are several projects on reconfigurable arithmetic arrays as mentioned in Section 1, to our knowledge there are no solutions similar to Function Folding in the literature. The following architectures differ considerably from our approach, but also allow to quickly change the operations performed by a PE.

The RAW microprocessor [8] also contains a cluster of processing elements, but they are rather complex processors. Therefore their programs cannot be gen- erated automatically as easily as the Function Folding PE programs. The RAW architecture resembles more a multiprocessor on a chip.

On the other hand, the MorphoSys architecture [5] follows a SIMD approach. All PEs in a row or column are controlled by a (global) program and execute the same instruction. This makes the PEs simpler, but the SIMD principle considerably restricts the available computations which can be executed. The array is also much harder to program.

Finally, *multi-context devices* provide two or more complete configuration contexts [9]. This technique is adapted from multi-context FPGAs. However, it does not allow frequent reconfigurations since the shadow configurations first have to be loaded completely. The configurations are completely independent. Multi-context devices hide the configuration latency to a cer- tain extent, but do not overcome the general efficiency problems of coarse-grain reconfigurable architectures.

7 Conclusions and Future Work

We have presented the architecture and function- ality of the Function Folding Processing Element for an enhanced PACT XPP- architecture. Preliminary analyses of both a hardware implementation and ap- plications mapped to this architecture indicate that Function Folding significantly increases the silicon ef- ficiency compared to current reconfigurable arithmetic arrays and has the potential to reduce the power con- sumption.

After the implementation of a Function Folding PE and the analysis of its parameters, future work will include the evaluation of paths which are not throughput-critical as mentioned at the end of Sec- tion 4.1. Integrated clustering and place-and-route al- gorithms will be explored. We also consider develop- ing a direct compiler from C to Function Folding PEs which might exploit their capabilities better then the current design flow via NML.

References

[1] R. Hartenstein. A decade of reconfigurable com- puting: a visionary retrospective. In *Proc. Design, Automation and Teat in Europe,* 2001.

[2] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. Architecture de- sign of reconfigurable pipelined datapaths. In *Proc. 20th Anniversary Conference on Advanced Research in VLSI,* Atlanta, GA, March 1999.

[3] R. Hartenstein, R. Kress, and H. Reinig. A new FPGA architecture for word-oriented datap- aths. In *Proc. Field-Programmabk Logic; 4th In- ternational Workshop.* Springer- Verlag, September 1994.

[4] V. Baumgarte, G- Ehlers, F. May, A. Nuckel, M. Vorbach, and M. Weinhardt. PACT XPP - a self-reconfigurable data processing architecture. *The Journal of Svpercompuiing,* 26(2), September 2003.

[5] M.-H. Lee, H. Singh, G. Lu, N. Bagherzadeh, and F. J. Kurdahi. Design and implementation

of the MorphoSys reconfigurable computing processor. *Journal of VLSI and Signal Processing-Systems for Signal, Image and Video Technology,* March 2000. March 2000.

[6] T. Stansfield. Using multiplexers for control and data in D-Fabrix. In *Field Programmable Logic and Applications,* LNCS 2778, pages 416-425. Springer, 2003.

[7] T. J. Cailahan, P. Chong, A. DeHon, and J. Wawrzynek. Fast module mapping and placement for datapaths in FPGAs. In *Proc FPGA '98,* Monterrey, CA, 1998.

[8] M. B. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro,* March/ April 2002.

[9] B. Salefski and L. Caglar. Re-configurable computing in wireless. In *Proc. 38th Design Automation Conference,* Las Vegas, NV, June 2001.

Claims

1. A data processing device comprising a multidimensional array of coarse grained logic elements (PAEs) processing data and operating at a first clock rate and communicating with one another -aird/or other elements via busses and/or communication lines operated at a second clock rate,
wherein the first clock rate is higher than the second and wherein the coarse grained logic elements comprise storage me¬ ans for storing data needed to be processed.

2. A data processing device according to claim 1 wherein the data processing of the array is controlled in a data-flow-like manner.

3. A data processing device according to claim 2, wherein the data storage means is adapted for storage of operands and/or intermediate results and wherein a valid bit is provided for each entry.

4. A data processing device according to a previous claim wherein data processing of a coarse grained logic element of the array is adapted to be effected in response to all valid bits of data and/or triggers needed being valid.

5. A processing array in particular according to a previous claim having a main data flow direction, said processing array having coarse grained logic elements and said coarse grained logic elements being adapted to effect data proces¬ sing while allowing data to flow in said in one direction,

_in particular ALUs having an upstream input side and a data downstream outrput side wherein at least some of said coar¬se grained logic elements- have data processing means such as second ALUs allowing data flow in a reverse direction.

6. A processing array according to the previous claim wherein a the instruction set for the ALUs in one direction is diffe¬rent from the instruction set of the ALUs in-'the reverse di¬rection.

7 .A processing array according to tone of the two previous claims wherein at least one coarse grained logic element comprises an ALU in one direction and an ALU in the reverse direction.

8. A processing device wherein the coarse grained element is connected to the busses and rows of coarse grained ele¬ments are provided interconnected via busses, wherein at least one input is connected to an upper row and at least one input is connected to a row below the cell and/or where this holds for an output connect.

9. A processing device according to the previous claim wherein the coarse grained element is connected to the busses and at least two input/output bus connects are provided in one row and wherein a switch in the bus structure and/or a gate or buffer or multiplexer is pro¬vided in the segment between inpunt and/or output.

10. A method of routing a processing array adapted to auto¬matically connect separated fragments of a configuration and /or configurations and to rip up nonconnectable tra¬ces in a stepwise manner.

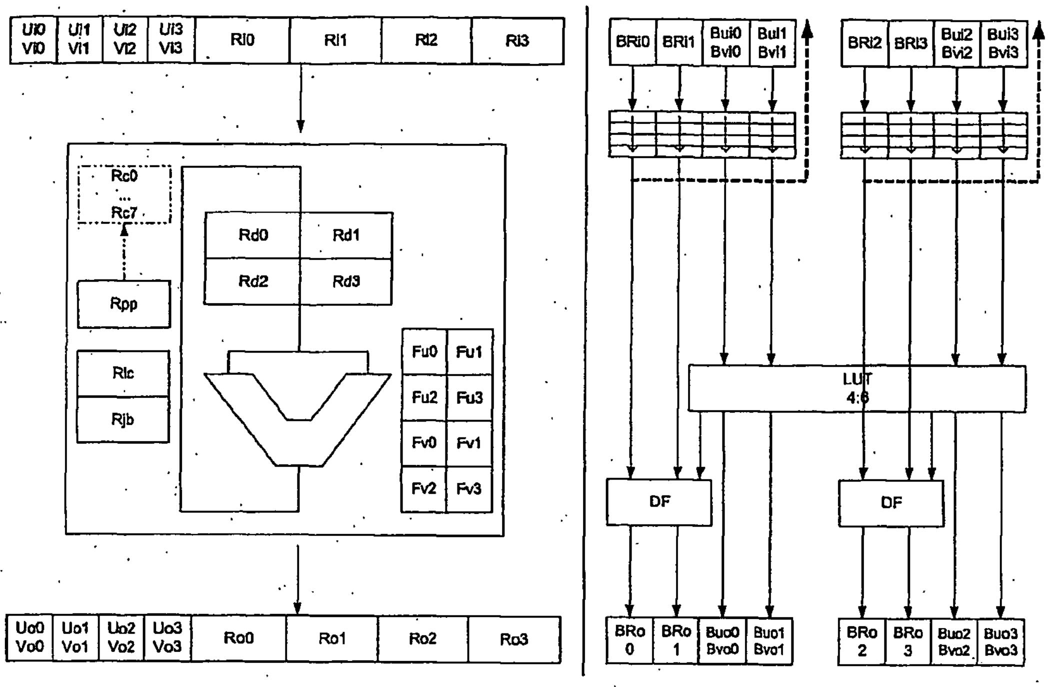

Ui0 Vi0
Ui1 Vi1
Ui2 Vi2
Ui3 Vi3
Ri0
Ri1
Ri2
Ri3
Rc0 ... Rc7
Rpp
Ric
Rjb
Rd0
Rd1
Rd2
Rd3
Fu0
Fu1
Fu2
Fu3
Fv0
Fv1
Fv2
Fv3
Uo0 Vo0
Uo1 Vo1
Uo2 Vo2
Uo3 Vo3
Ro0
Ro1
Ro2
Ro3
BRi0
BRi1
Bui0 Bvi0
Bui1 Bvi1
BRi2
BRi3
Bui2 Bvi2
Bui3 Bvi3
LUT 4:6
DF
DF
BRo 0
BRo 1
Buo0 Bvo0
Buo1 Bvo1
BRo 2
BRo 3
Buo2 Bvo2
Buo3 Bvo3

Fig. 1

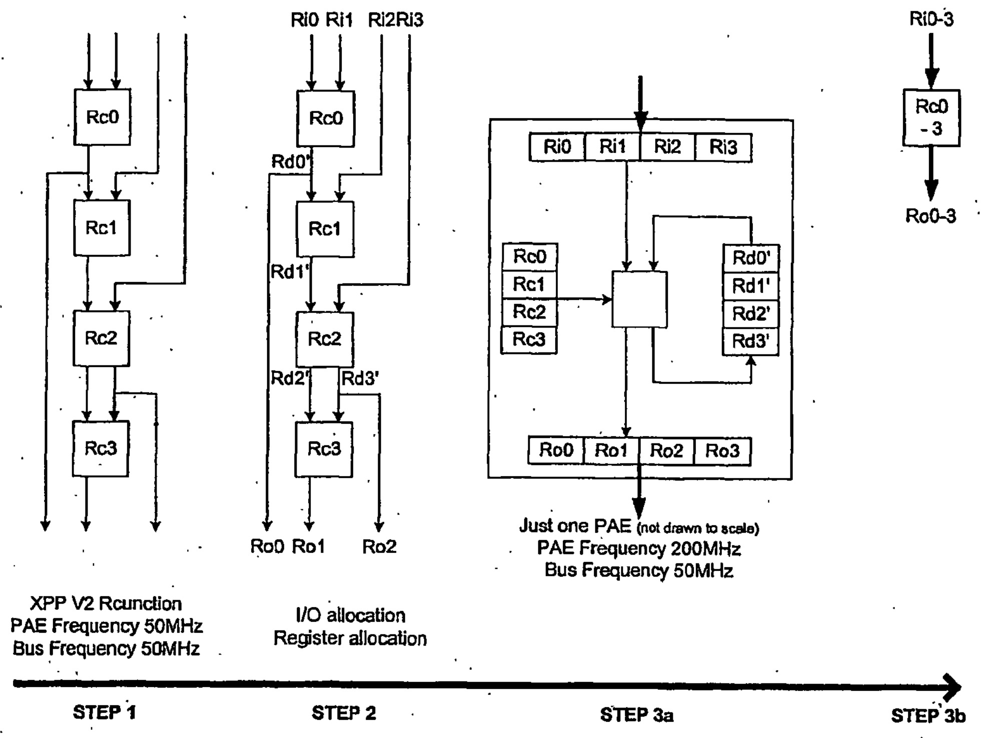

Rc0
Rc1
Rc2
Rc3
XPP V2 Rcunction
PAE Frequency 50MHz
Bus Frequency 50MHz
Ri0 Ri1 Ri2Ri3
Rd0'
Rd1'
Rd2'
Rd3'
Ro0 Ro1 Ro2
I/O allocation
Register allocation
Ri0 Ri1 Ri2 Ri3
Rc0 Rc1 Rc2 Rc3
Rd0' Rd1' Rd2' Rd3'
Ro0 Ro1 Ro2 Ro3
Just one PAE (not drawn to scale)
PAE Frequency 200MHz
Bus Frequency 50MHz
Ri0-3
Rc0 -3
Ro0-3
STEP 1
STEP 2
STEP 3a
STEP 3b

Fig. 2

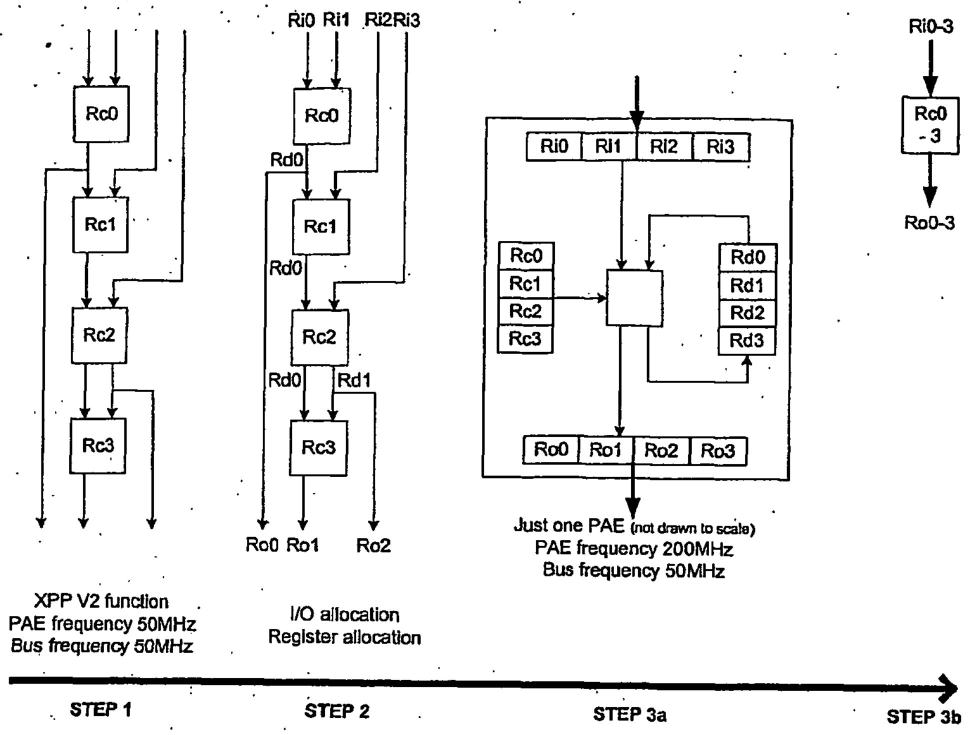

Rc0
Rc1
Rc2
Rc3
Ri0 Ri1 Ri2Ri3
Rd0
Rd1
Ro0 Ro1 Ro2
Ri0 Ri1 Ri2 Ri3
Rd0 Rd1 Rd2 Rd3
Ro0 Ro1 Ro2 Ro3
Just one PAE (not drawn to scale)
PAE frequency 200MHz
Bus frequency 50MHz
Ri0-3
Rc0 - 3
Ro0-3
XPP V2 function
PAE frequency 50MHz
Bus frequency 50MHz
I/O allocation
Register allocation
STEP 1
STEP 2
STEP 3a
STEP 3b

Fig. 3

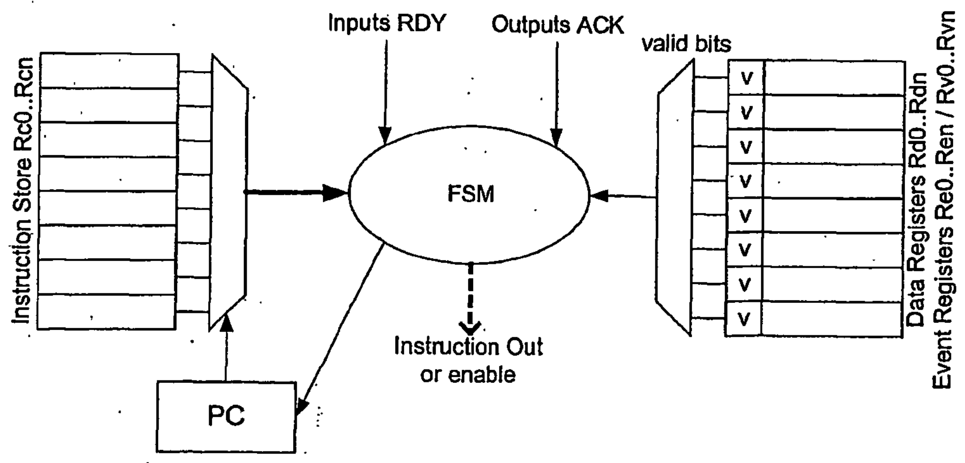

Instruction Store Rc0..Rcn
Inputs RDY
Outputs ACK
valid bits
FSM
V
Instruction Out or enable
PC
Data Registers Rd0..Rdn
Event Registers Re0..Ren / Rv0..Rvn

Fig. 4

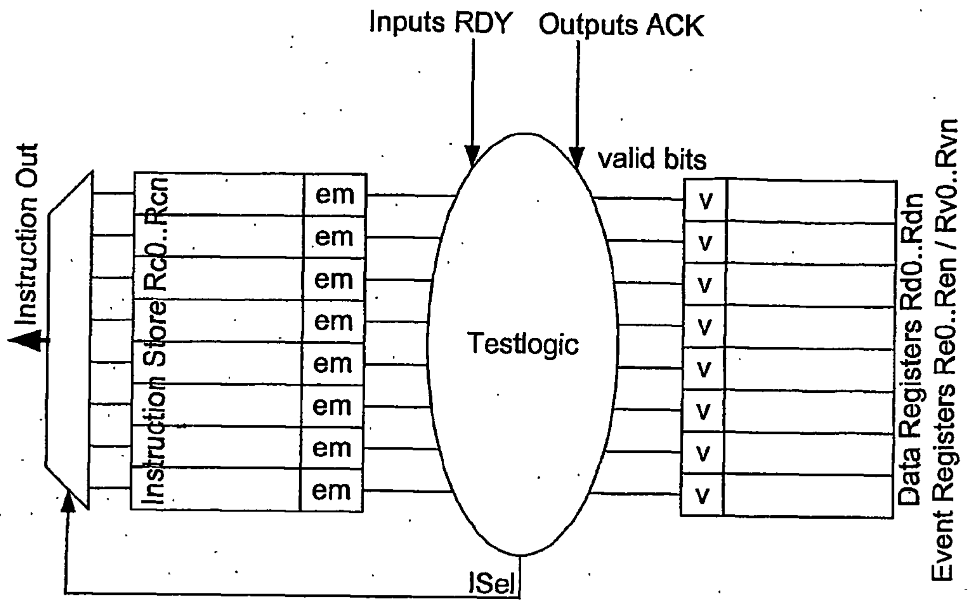

Inputs RDY
Outputs ACK
valid bits
Instruction Out
Instruction Store Rc0..Rcn
em
Testlogic
v
Data Registers Rd0..Rdn
Event Registers Re0..Ren / Rv0..Rvn
ISel

Fig. 5

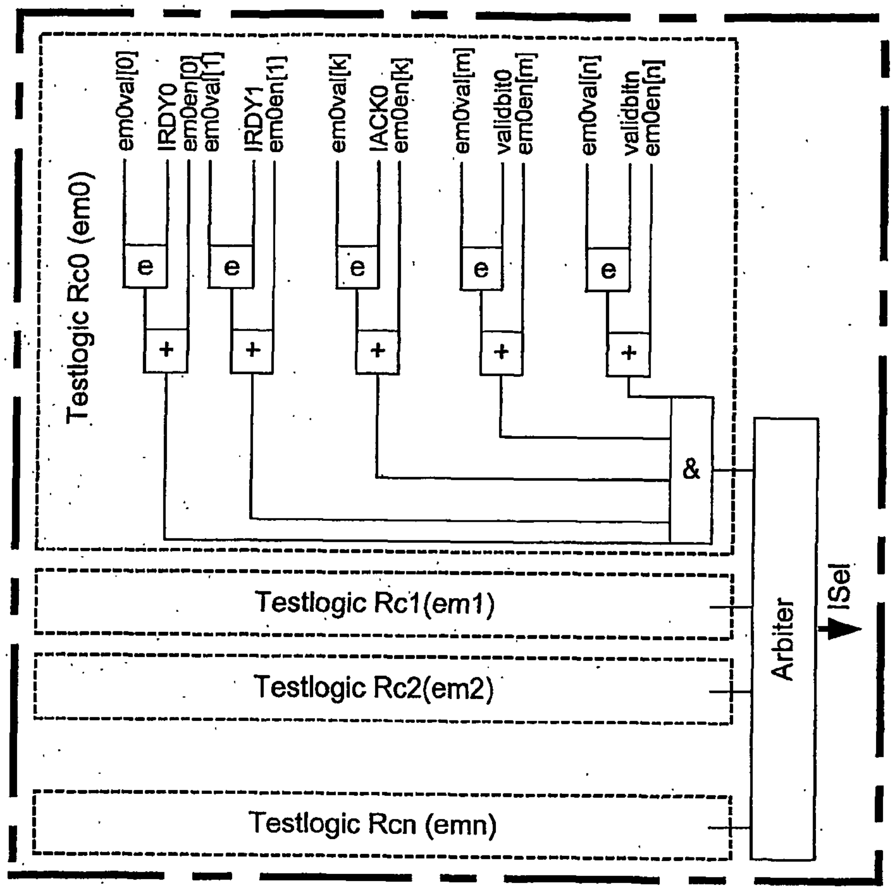

Testlogic Rc0 (em0)
em0val[0]
IRDY0
em0en[0]
em0val[1]
IRDY1
em0en[1]
em0val[k]
IACK0
em0en[k]
em0val[m]
validbit0
em0en[m]
em0val[n]
validbitn
em0en[n]
e
+
&
Testlogic Rc1(em1)
Testlogic Rc2(em2)
Testlogic Rcn (emn)
Arbiter
ISel

Fig. 6

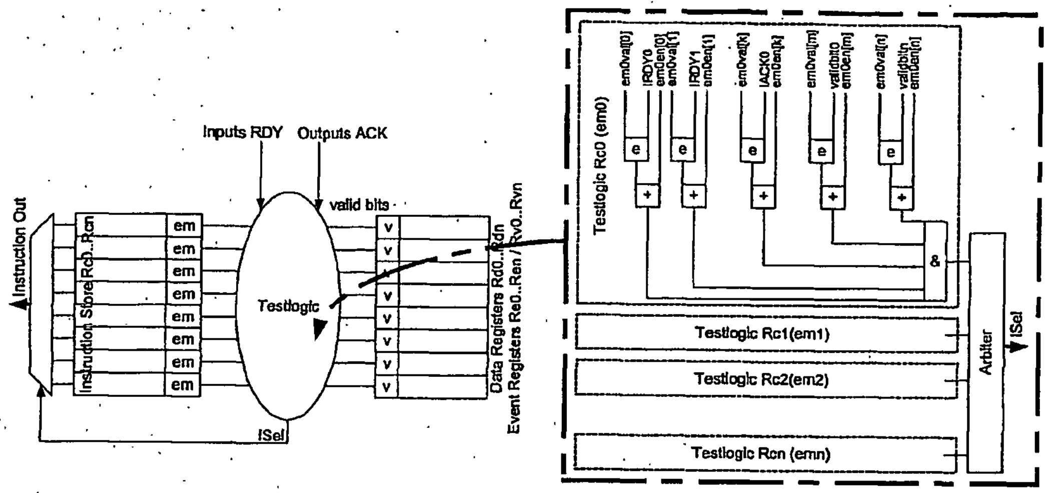

Inputs RDY
Outputs ACK
valid bits
Instruction Out
Instruction Store Rc0..Rcn
em
Testlogic
v
Data Registers Rd0..Rdn
Event Registers Re0..Ren / Rv0..Rvn
ISel
Testlogic Rc0 (em0)
em0val[0]
IRDY0
em0en[0]
em0val[1]
IRDY1
em0en[1]
em0val[k]
IACK0
em0en[k]
em0val[m]
validbit0
em0en[m]
em0val[n]
validbitn
em0en[n]
e
+
&
Testlogic Rc1(em1)
Testlogic Rc2(em2)
Testlogic Rcn (emn)
Arbiter
ISel

Fig. 7

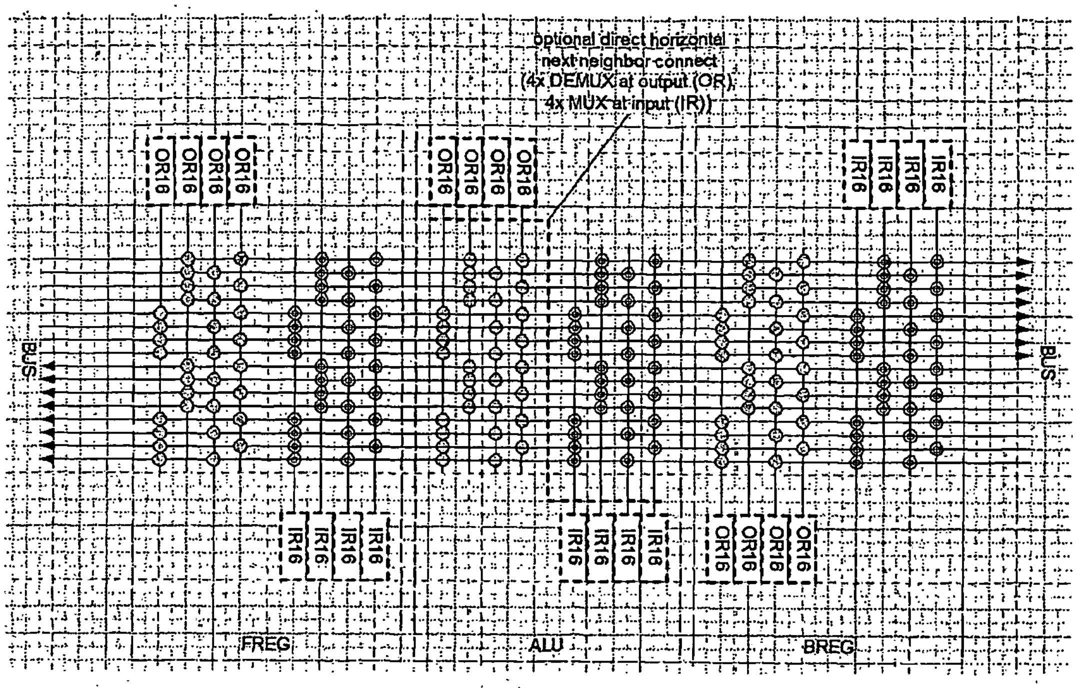

optional direct horizontal next neighbor connect (4x DEMUX at output (OR), 4x MUX at input (IR))
OR16
IR16
BUS
FREG
ALU
BREG

Fig. 8

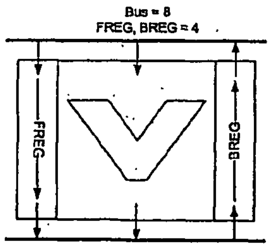

Bus = 8
FREG, BREG = 4
FREG
BREG

Fig. 9

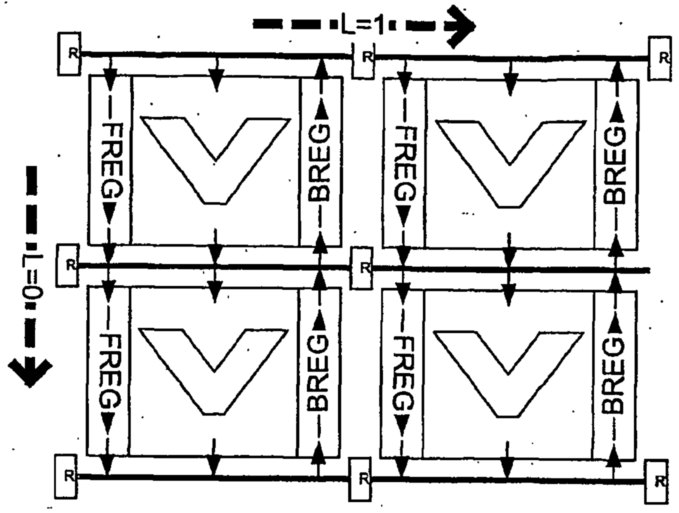


Fig. 10

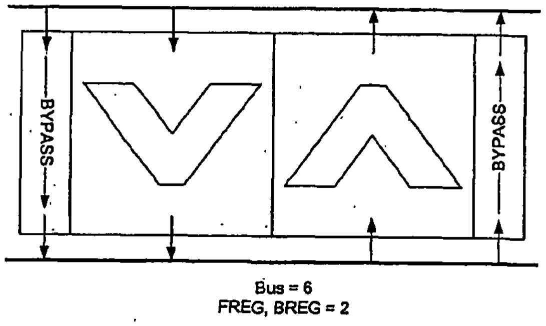


Fig. 11

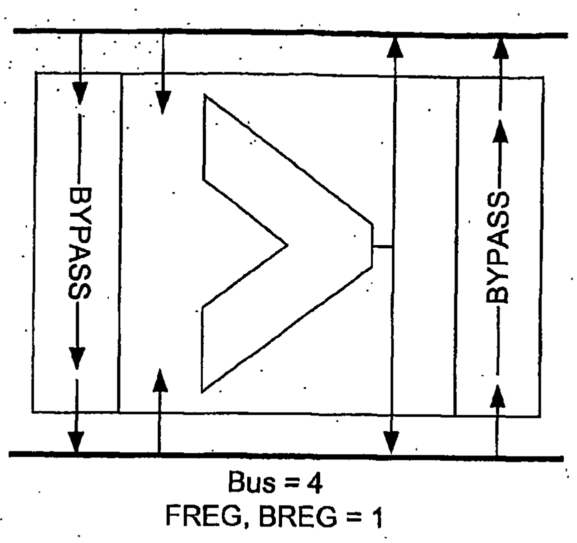


Fig. 12

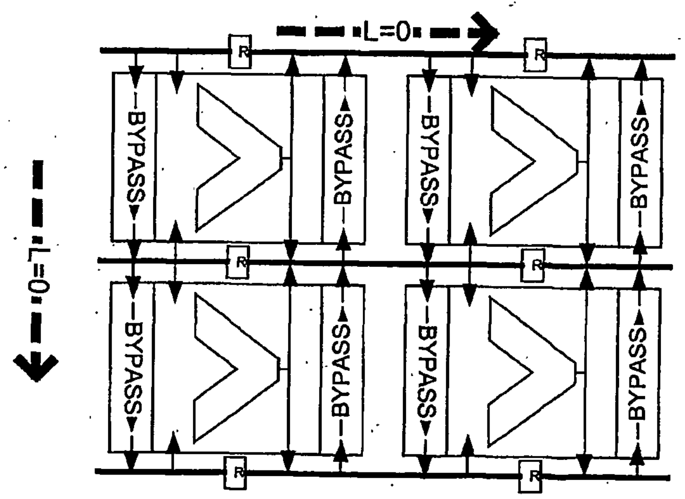


Fig. 13

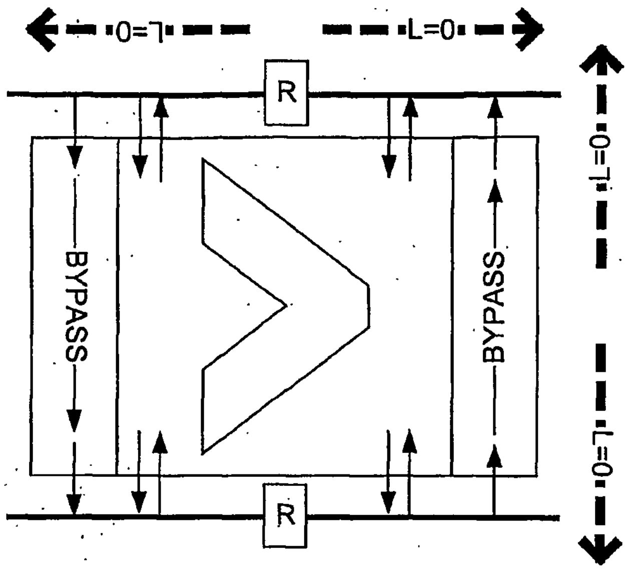

L=0
L=0
L=0
L=0
R
R
BYPASS
BYPASS

Fig. 14

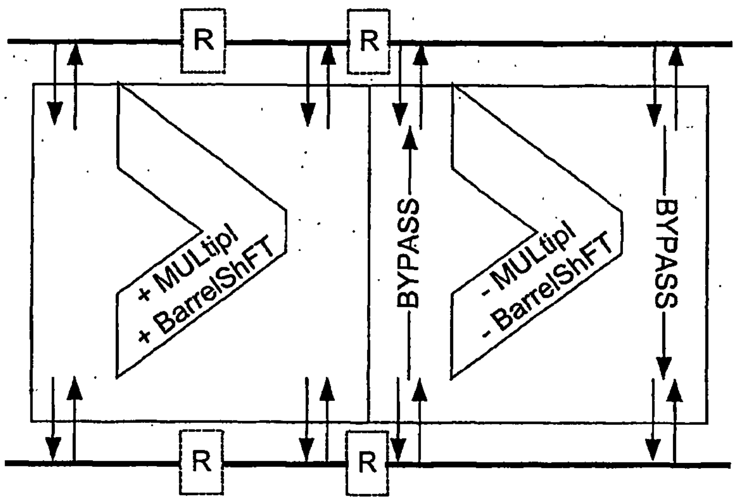
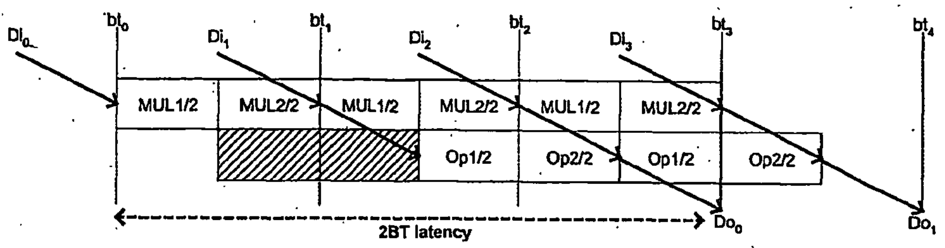
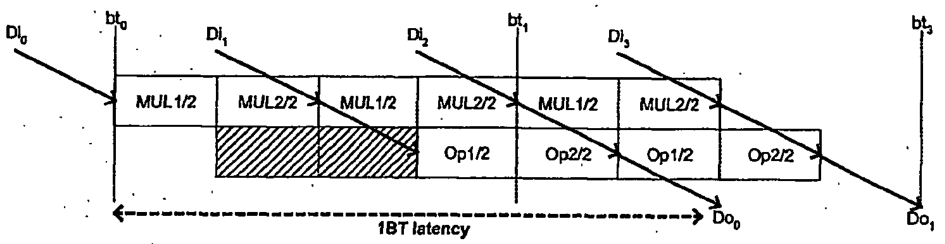

R
R
R
R
+ MULtipl
+ BarrelShFT
BYPASS
- MULtipl
- BarrelShFT
BYPASS

Fig. 15

FF = 2:



FF = 4:



FF>= 6:

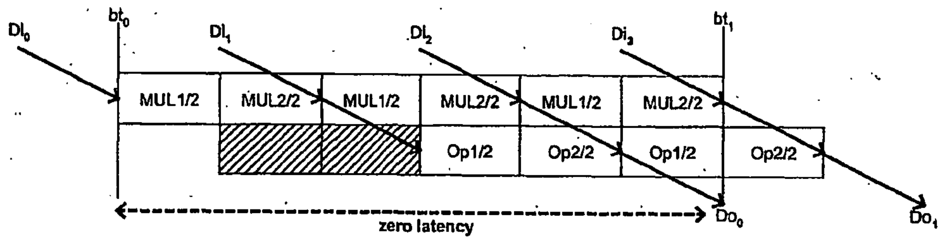


Fig. 16

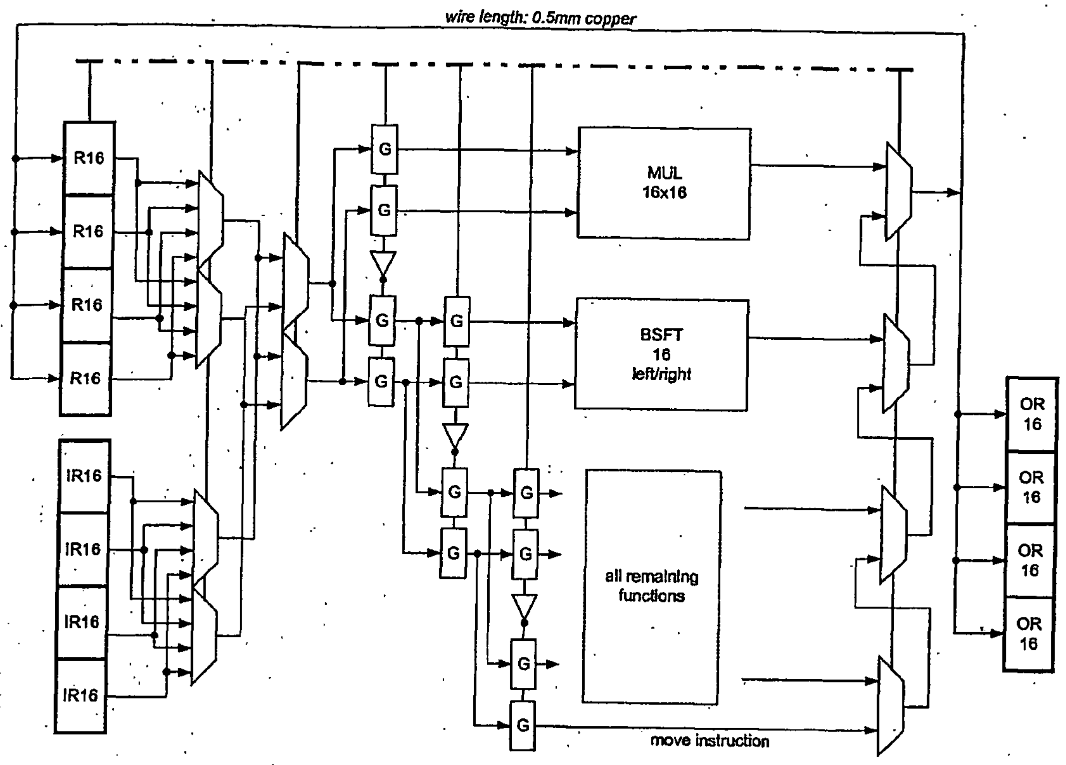

wire length: 0.5mm copper
R16
R16
R16
R16
IR16
IR16
IR16
IR16
G
MUL 16x16
BSFT 16 left/right
all remaining functions
move instruction
OR 16
OR 16
OR 16
OR 16

Fig. 17

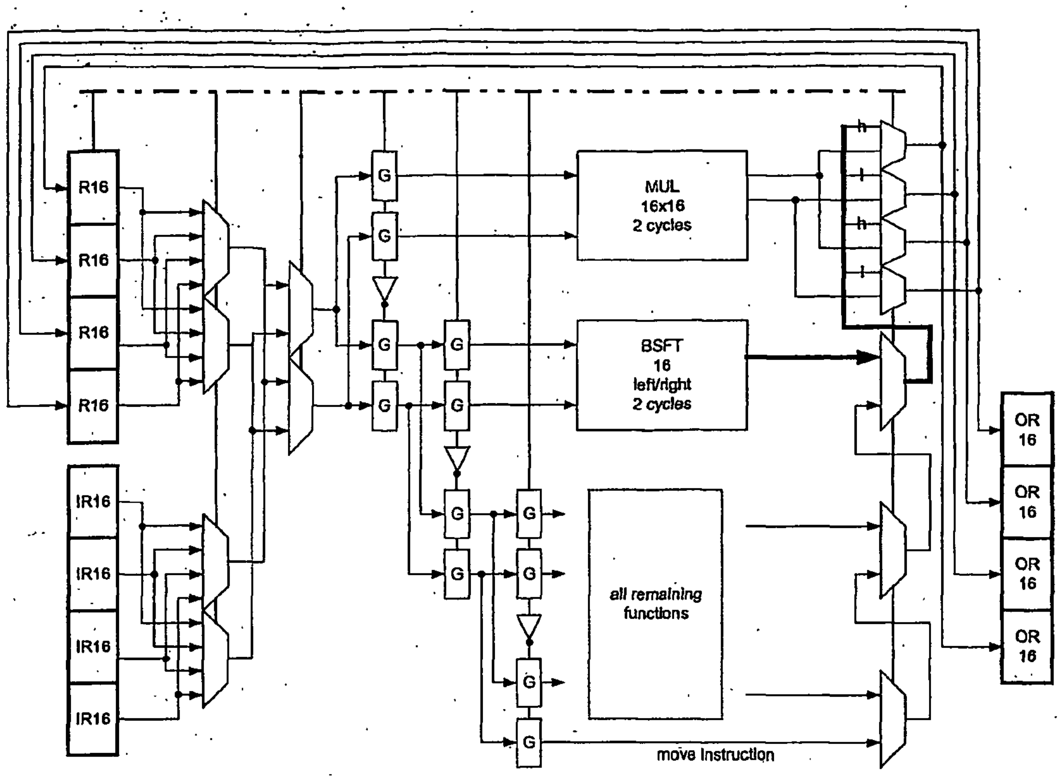


Fig. 18

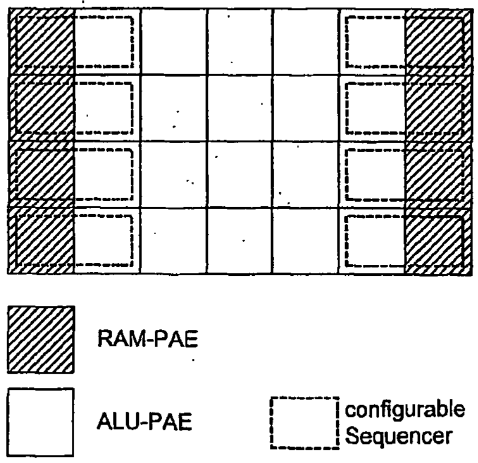


Fig. 19: Configurable Sequencer

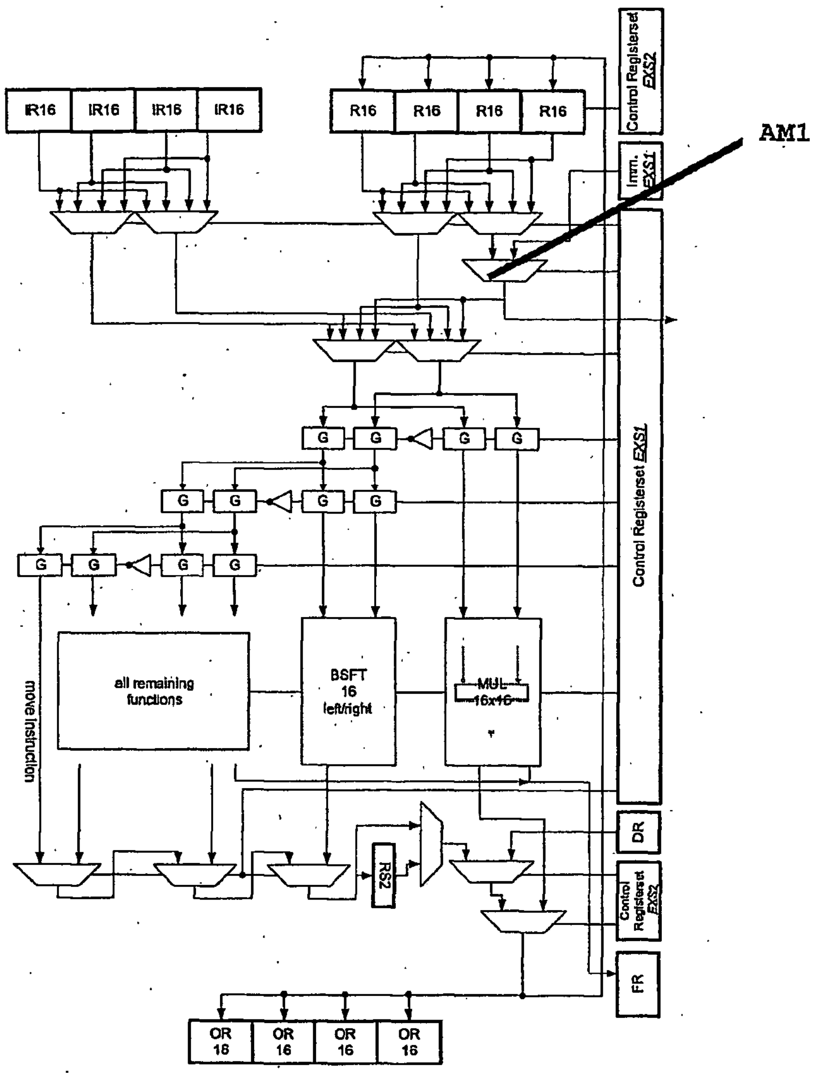


Figure 20: Enhanced Version of the ALU-PAE

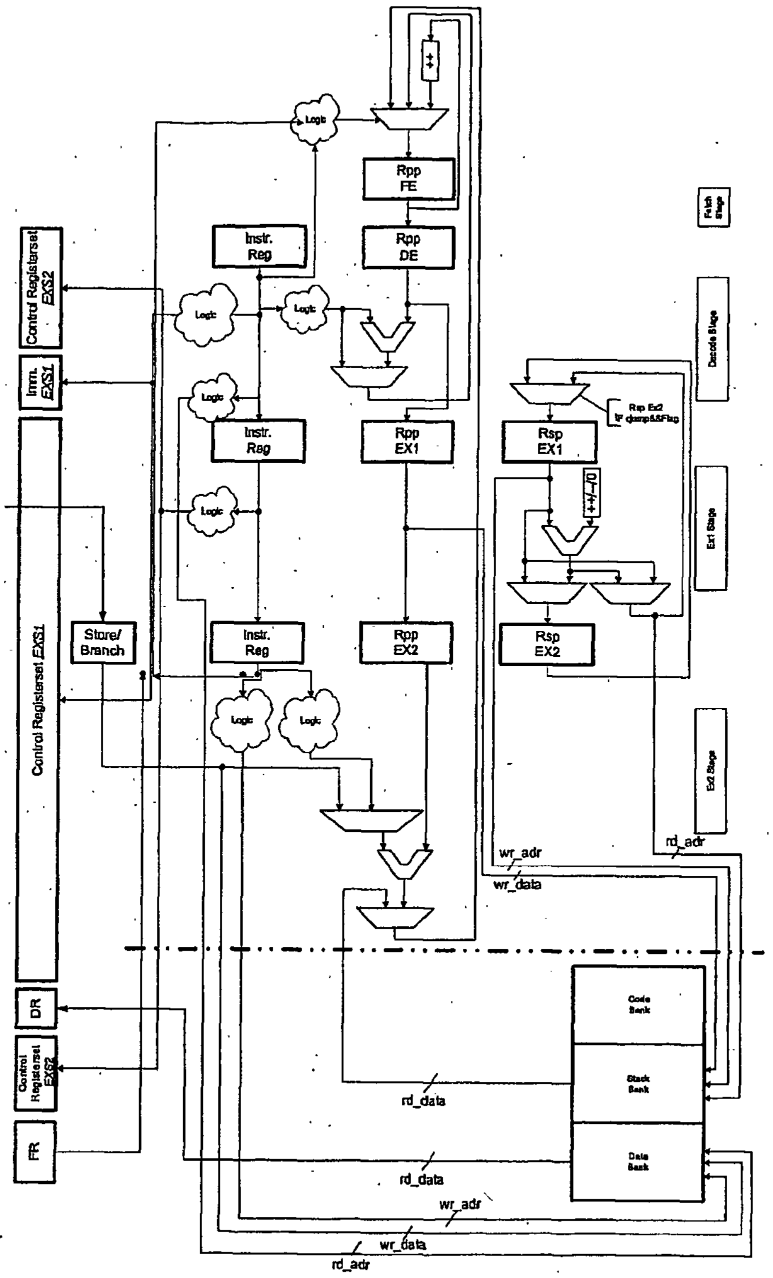


Figure 21: Overview of the RAM-PAE

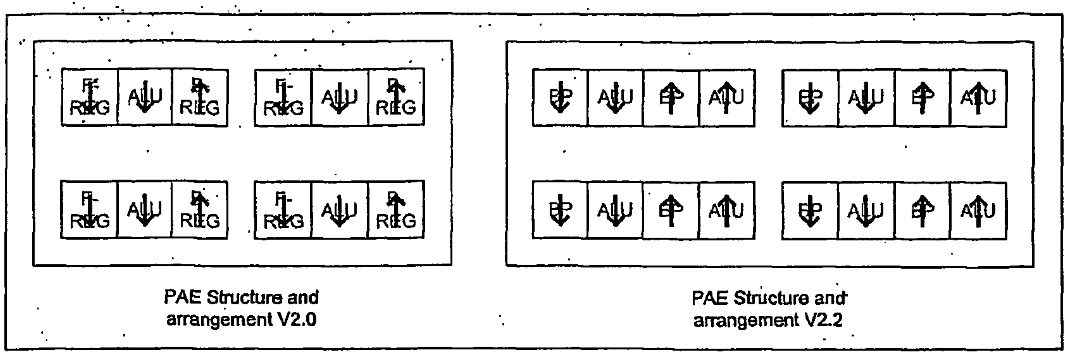

FREG
ALU
BREG
EP
PAE Structure and arrangement V2.0
PAE Structure and arrangement V2.2

Fig. 22

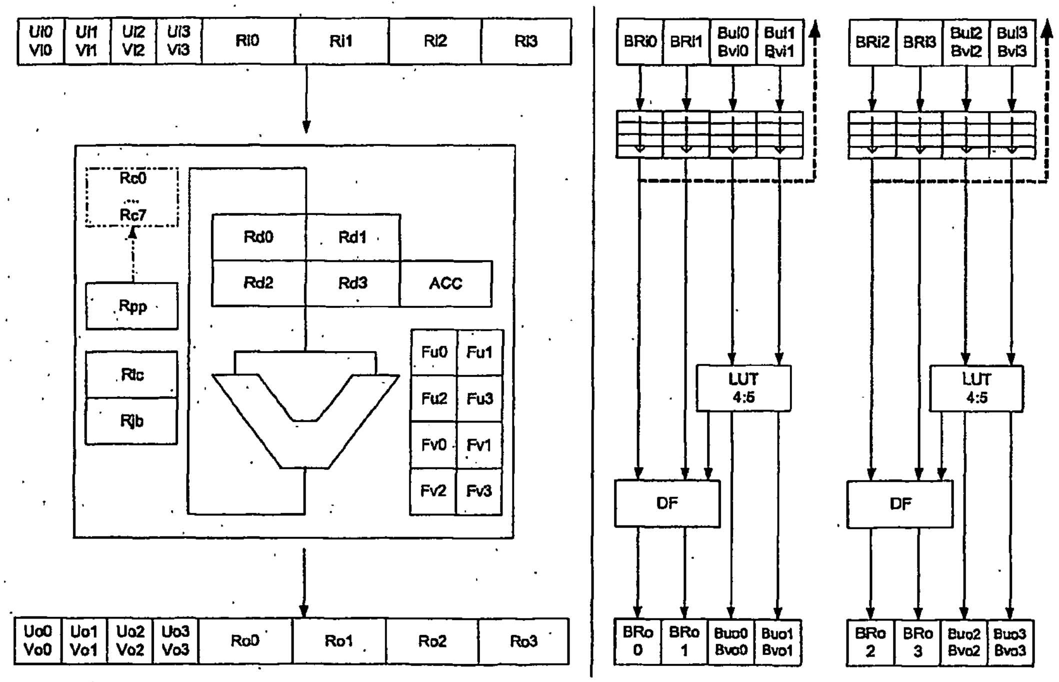

Ui0 Vi0
Ui1 Vi1
Ui2 Vi2
Ui3 Vi3
Ri0
Ri1
Ri2
Ri3
Rc0 ... Rc7
Rpp
Rlc
Rjb
Rd0
Rd1
Rd2
Rd3
ACC
Fu0
Fu1
Fu2
Fu3
Fv0
Fv1
Fv2
Fv3
Uo0 Vo0
Uo1 Vo1
Uo2 Vo2
Uo3 Vo3
Ro0
Ro1
Ro2
Ro3
BRi0
BRi1
Bui0 Bvi0
Bui1 Bvi1
BRi2
BRi3
Bui2 Bvi2
Bui3 Bvi3
LUT 4:5
DF
BRo 0
BRo 1
Buo0 Bvo0
Buo1 Bvo1
BRo 2
BRo 3
Buo2 Bvo2
Buo3 Bvo3

Fig. 23

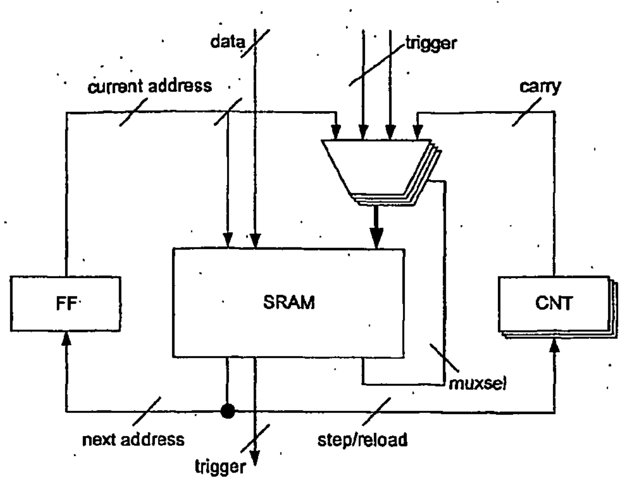

data
trigger
current address
carry
SRAM
FF
CNT
muxsel
next address
step/reload
trigger

Fig. 24

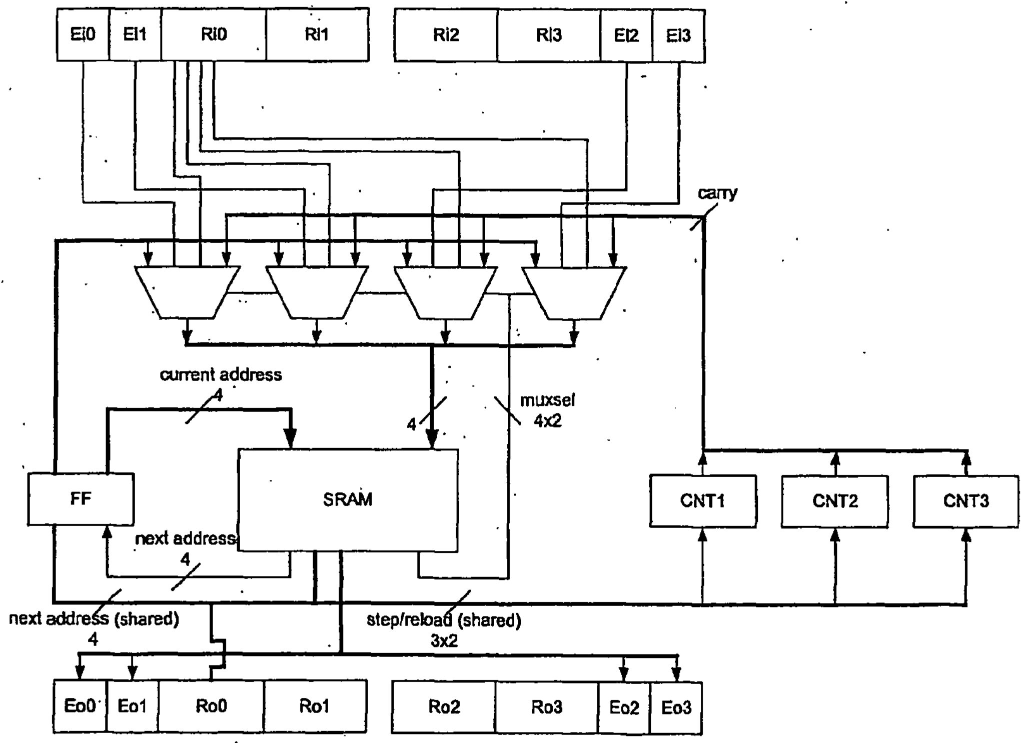

Ei0
Ei1
Ri0
Ri1
Ri2
Ri3
Ei2
Ei3
carry
current address
4
4
muxsel
4x2
FF
SRAM
CNT1
CNT2
CNT3
next address
4
next address (shared)
4
step/reload (shared)
3x2
Eo0
Eo1
Ro0
Ro1
Ro2
Ro3
Eo2
Eo3

Fig. 25

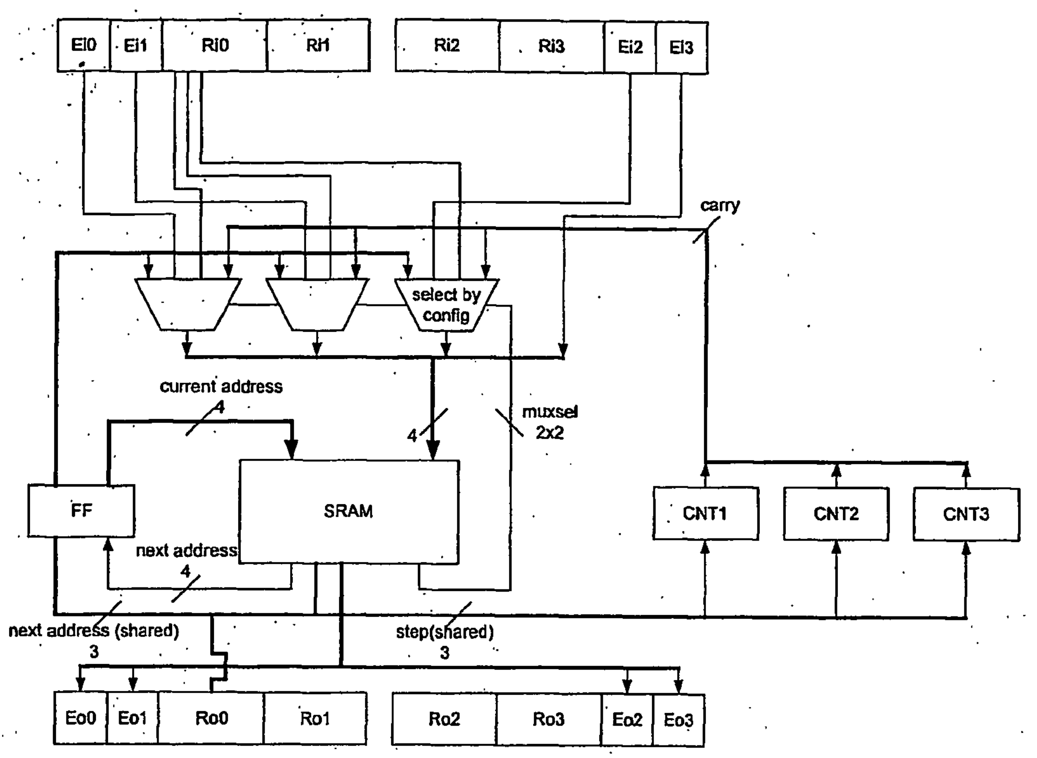

EI0
EI1
RI0
RI1
RI2
RI3
EI2
EI3
carry
select by config
current address
4
4
muxsel
2x2
FF
SRAM
CNT1
CNT2
CNT3
next address
4
next address (shared)
3
step(shared)
3
Eo0
Eo1
Ro0
Ro1
Ro2
Ro3
Eo2
Eo3

Fig. 26

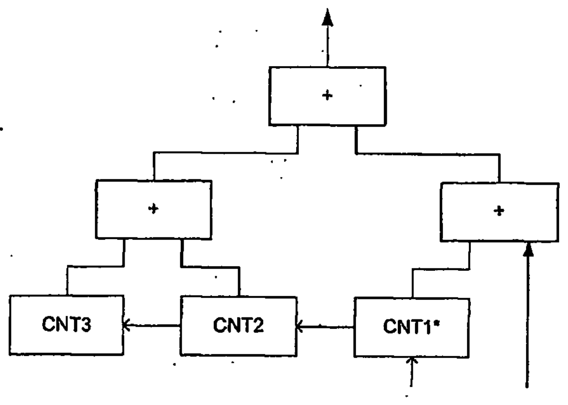

+
+
+
CNT3
CNT2
CNT1*

Fig. 27

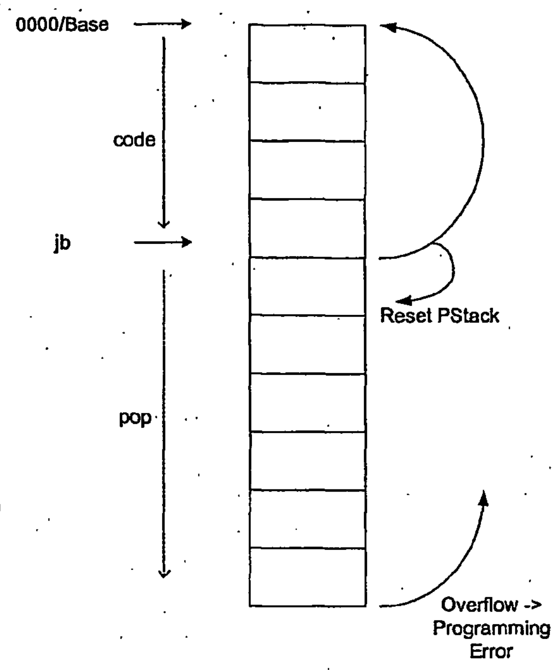

0000/Base
code
jb
Reset PStack
pop
Overflow -> Programming Error

Fig. 28

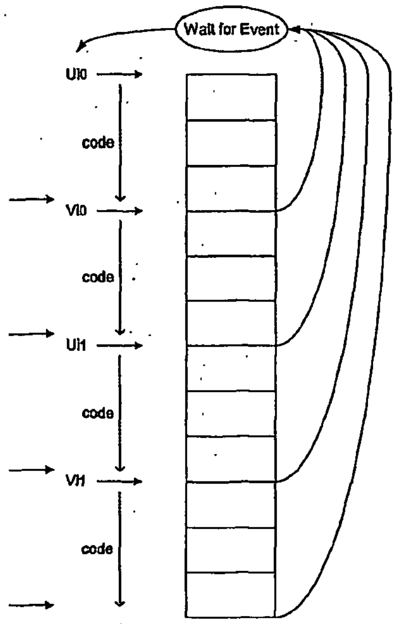

Wait for Event
Ui0
code
Vi0
code
Ui1
code
Vi1
code

Fig. 29

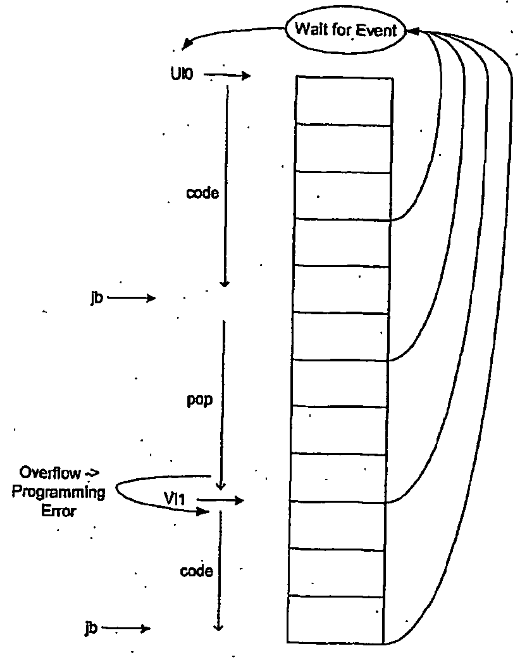

Wait for Event
UI0
code
jb
pop
Overflow -> Programming Error
VI1
code
jb

Fig. 30

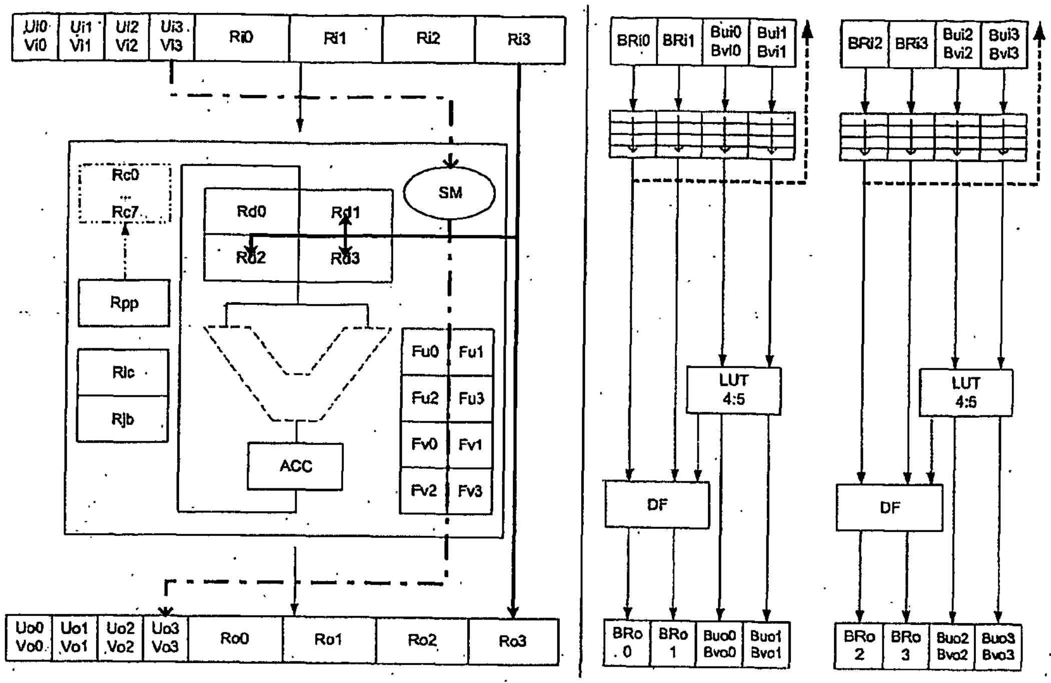

Ui0 Vi0
Ui1 Vi1
Ui2 Vi2
Ui3 Vi3
Ri0
Ri1
Ri2
Ri3
Rc0 ... Rc7
Rpp
Ric
Rjb
Rd0
Rd1
Rd2
Rd3
SM
ACC
Fu0
Fu1
Fu2
Fu3
Fv0
Fv1
Fv2
Fv3
Uo0 Vo0
Uo1 Vo1
Uo2 Vo2
Uo3 Vo3
Ro0
Ro1
Ro2
Ro3
BRi0
BRi1
Bui0 Bvi0
Bui1 Bvi1
BRi2
BRi3
Bui2 Bvi2
Bui3 Bvi3
LUT 4:5
LUT 4:5
DF
DF
BRo 0
BRo 1
Buo0 Bvo0
Buo1 Bvo1
BRo 2
BRo 3
Buo2 Bvo2
Buo3 Bvo3

Fig. 31

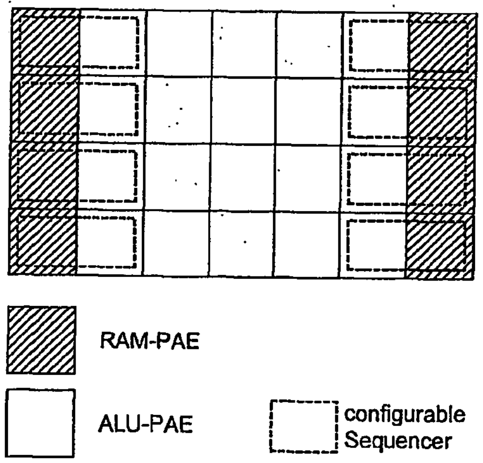
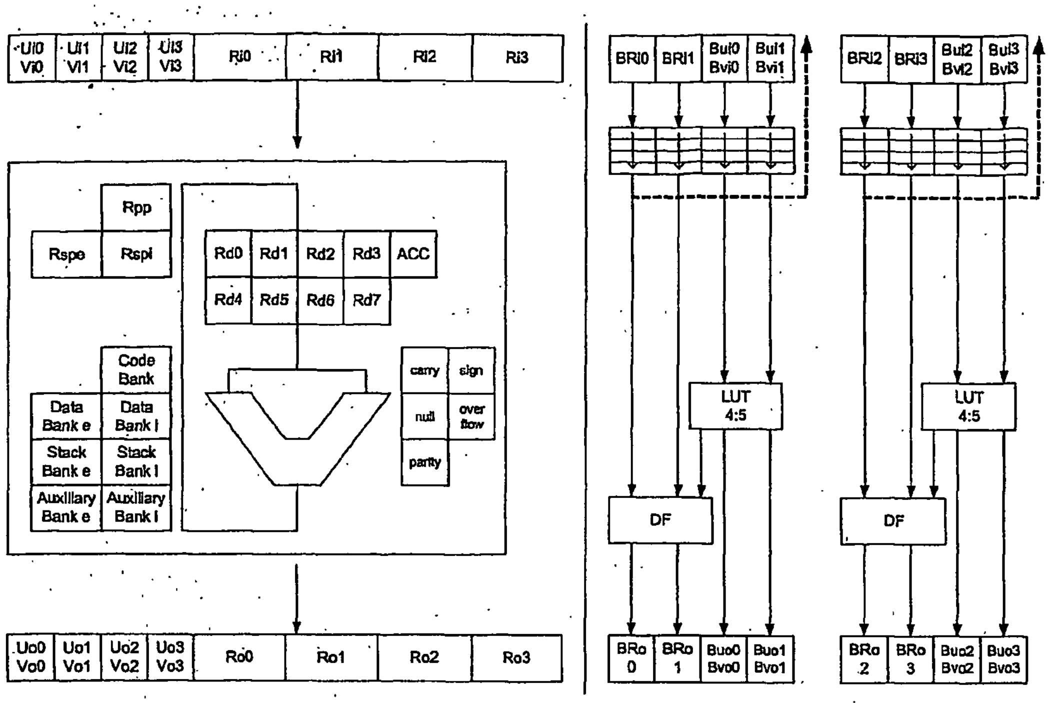

RAM-PAE
ALU-PAE
configurable Sequencer

Fig. 32



Ui0 Vi0
Ui1 Vi1
Ui2 Vi2
Ui3 Vi3
Ri0
Ri1
Ri2
Ri3
Rpp
Rspe
Rspi
Rd0
Rd1
Rd2
Rd3
ACC
Rd4
Rd5
Rd6
Rd7
Code Bank
Data Bank e
Data Bank i
Stack Bank e
Stack Bank i
Auxiliary Bank e
Auxiliary Bank i
carry
sign
null
over flow
parity
Uo0 Vo0
Uo1 Vo1
Uo2 Vo2
Uo3 Vo3
Ro0
Ro1
Ro2
Ro3
BRi0
BRi1
Bui0 Bvi0
Bui1 Bvi1
BRi2
BRi3
Bui2 Bvi2
Bui3 Bvi3
LUT 4:5
LUT 4:5
DF
DF
BRo 0
BRo 1
Buo0 Bvo0
Buo1 Bvo1
BRo 2
BRo 3
Buo2 Bvo2
Buo3 Bvo3

Fig. 33

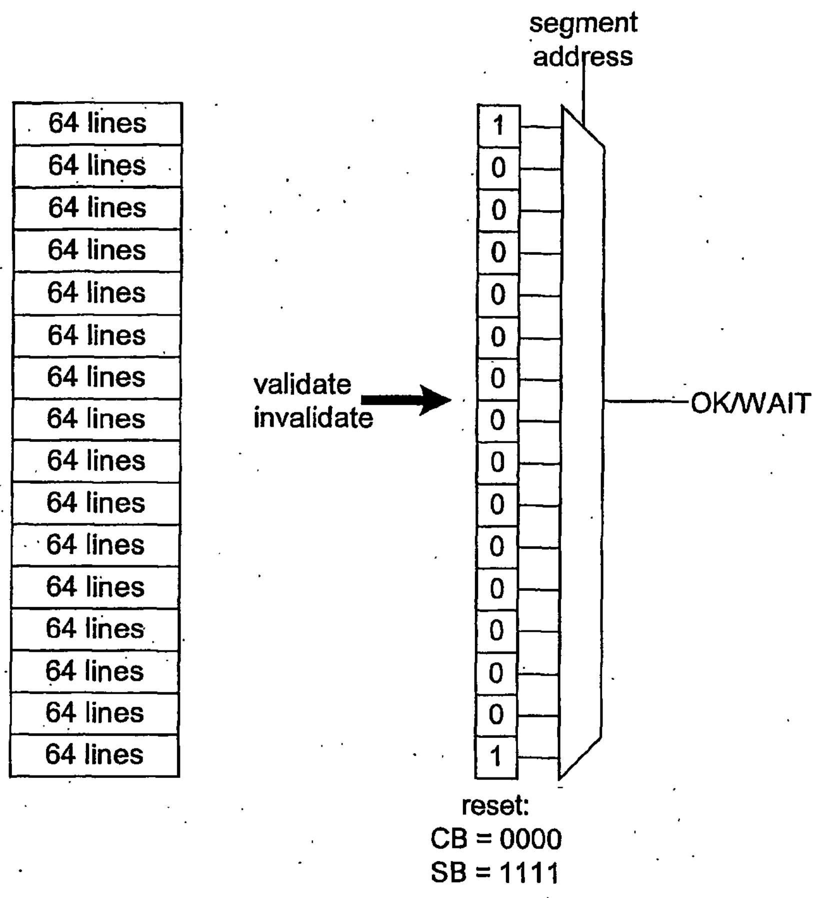

segment address
64 lines
64 lines
64 lines
64 lines
64 lines
64 lines
64 lines
64 lines
64 lines
64 lines
64 lines
64 lines
64 lines
64 lines
64 lines
64 lines
validate
invalidate
1
0
0
0
0
0
0
0
0
0
0
0
0
0
0
1
OK/WAIT
reset:
CB = 0000
SB = 1111

Fig. 34

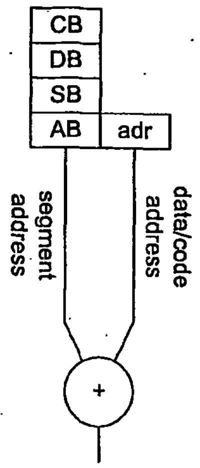

CB
DB
SB
AB
adr
segment address
data/code address
+

Fig. 35

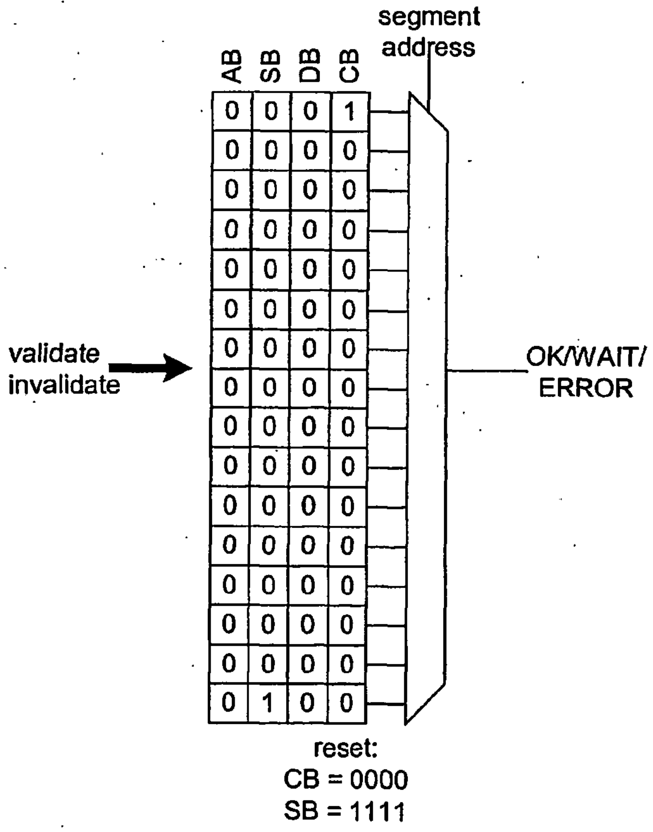
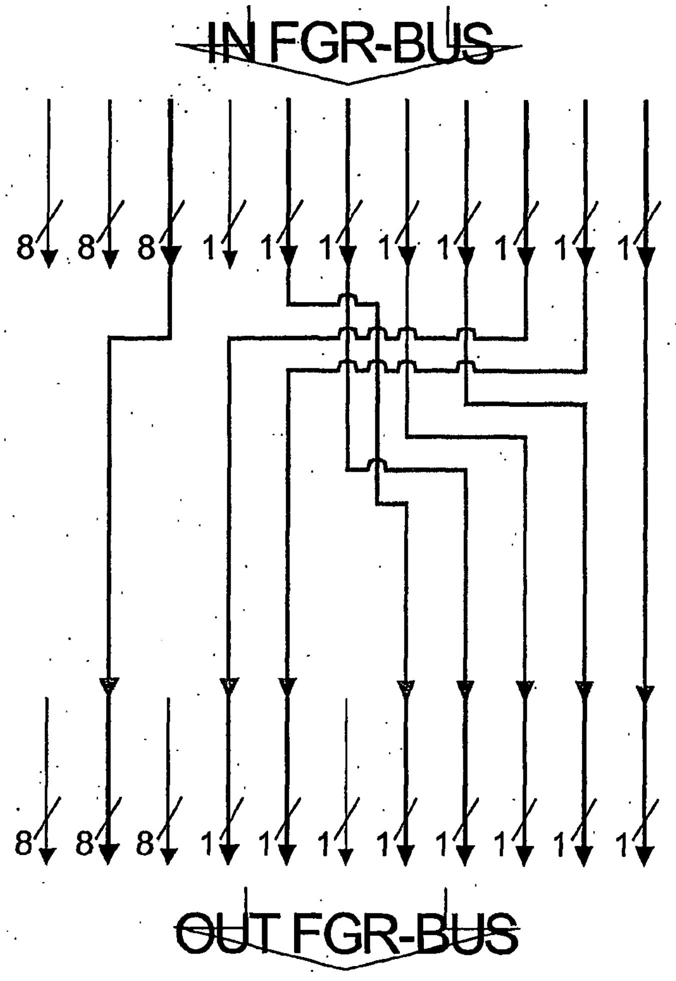

segment address
AB SB DB CB
0 0 0 1
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 1 0 0
validate invalidate
OK/WAIT/ ERROR
reset:
CB = 0000
SB = 1111

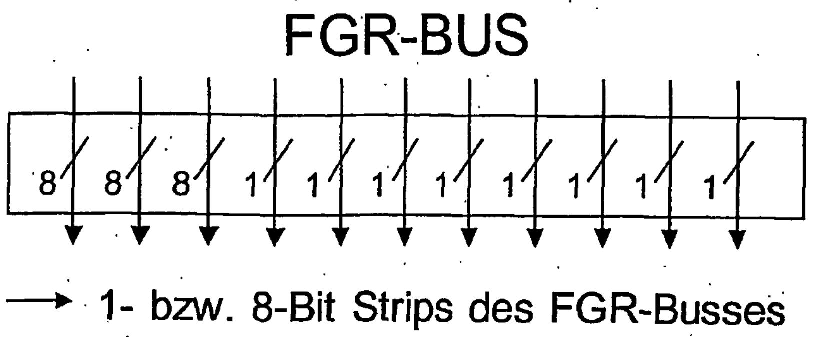
Fig. 36



→ freier Strip mit 1- bzw. 8-Bit Granularität
→ 4-Bit breite logische Verbindung
→ 10-Bit breite logische Verbindung
→ 1-Bit breite logische Verbindung
⇢ freier Strip mit 1- bzw. 8-Bit Granularität

(free strip of 1 or 8 bit granularity
4- Bit wide logic interconnection
10- Bit wide logic interconnection
free strip of 1 or 8 bit granularity)

Fig. 37



1 or 8 bit strip of FGR-Busses

Fig. 38

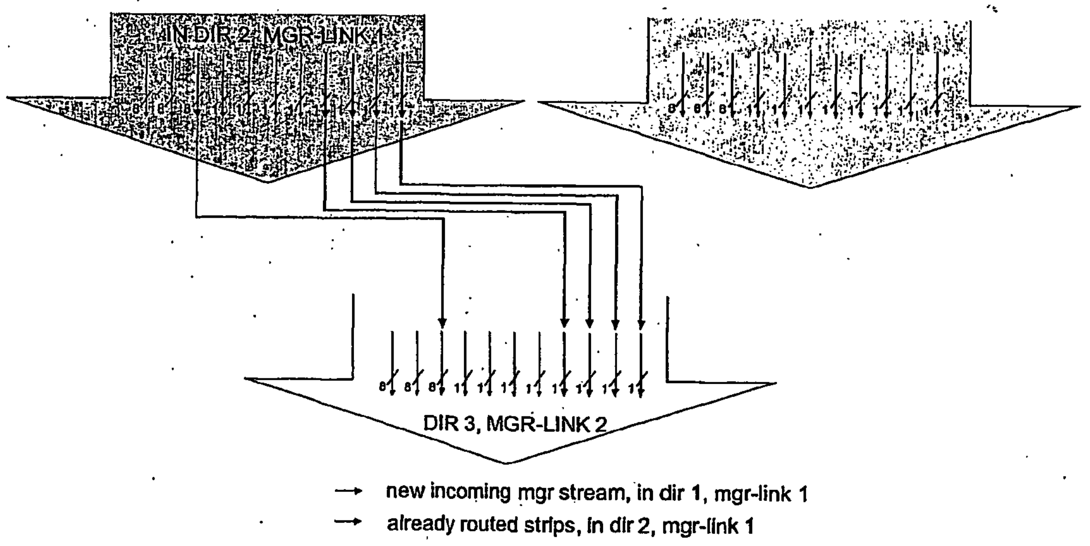

IN DIR 2, MGR-LINK 1
DIR 3, MGR-LINK 2
new incoming mgr stream, in dir 1, mgr-link 1
already routed strips, in dir 2, mgr-link 1

Fig. 39

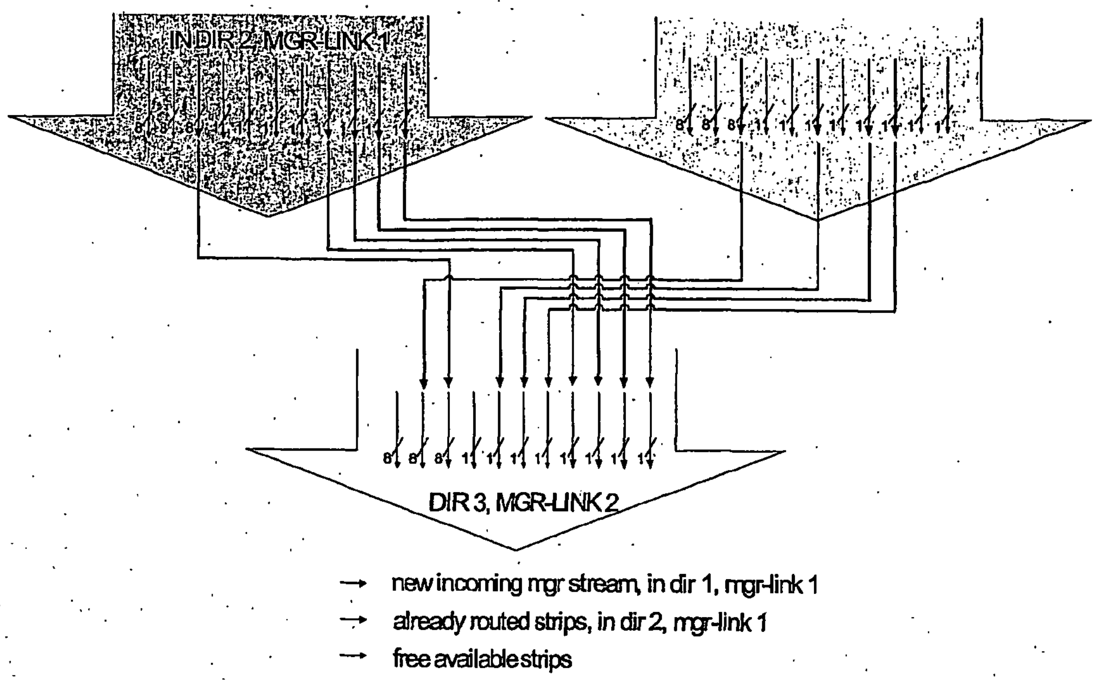

IN DIR 2, MGR-LINK 1
8 8 8 1 1 1 1 1 1 1 1
8 8 8 1 1 1 1 1 1 1 1
8 8 8 1 1 1 1 1 1 1 1
DIR 3, MGR-LINK 2
new incoming mgr stream, in dir 1, mgr-link 1
already routed strips, in dir 2, mgr-link 1
free available strips

Fig. 40

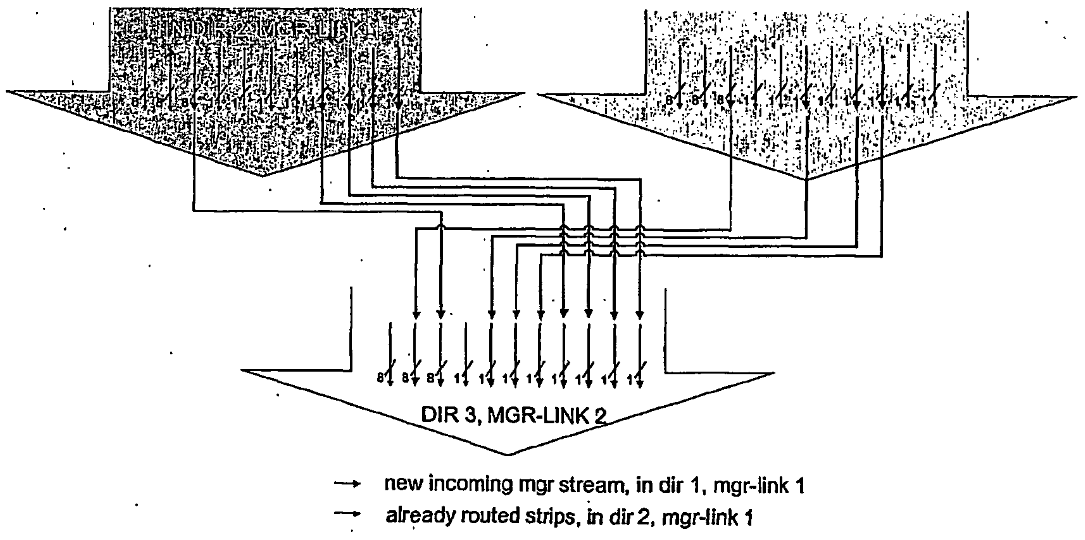

IN DIR 2, MGR-LINK 1
8 8 8 1 1 1 1 1 1 1 1
8 8 8 1 1 1 1 1 1 1 1
DIR 3, MGR-LINK 2
new incoming mgr stream, in dir 1, mgr-link 1
already routed strips, in dir 2, mgr-link 1

Fig. 41

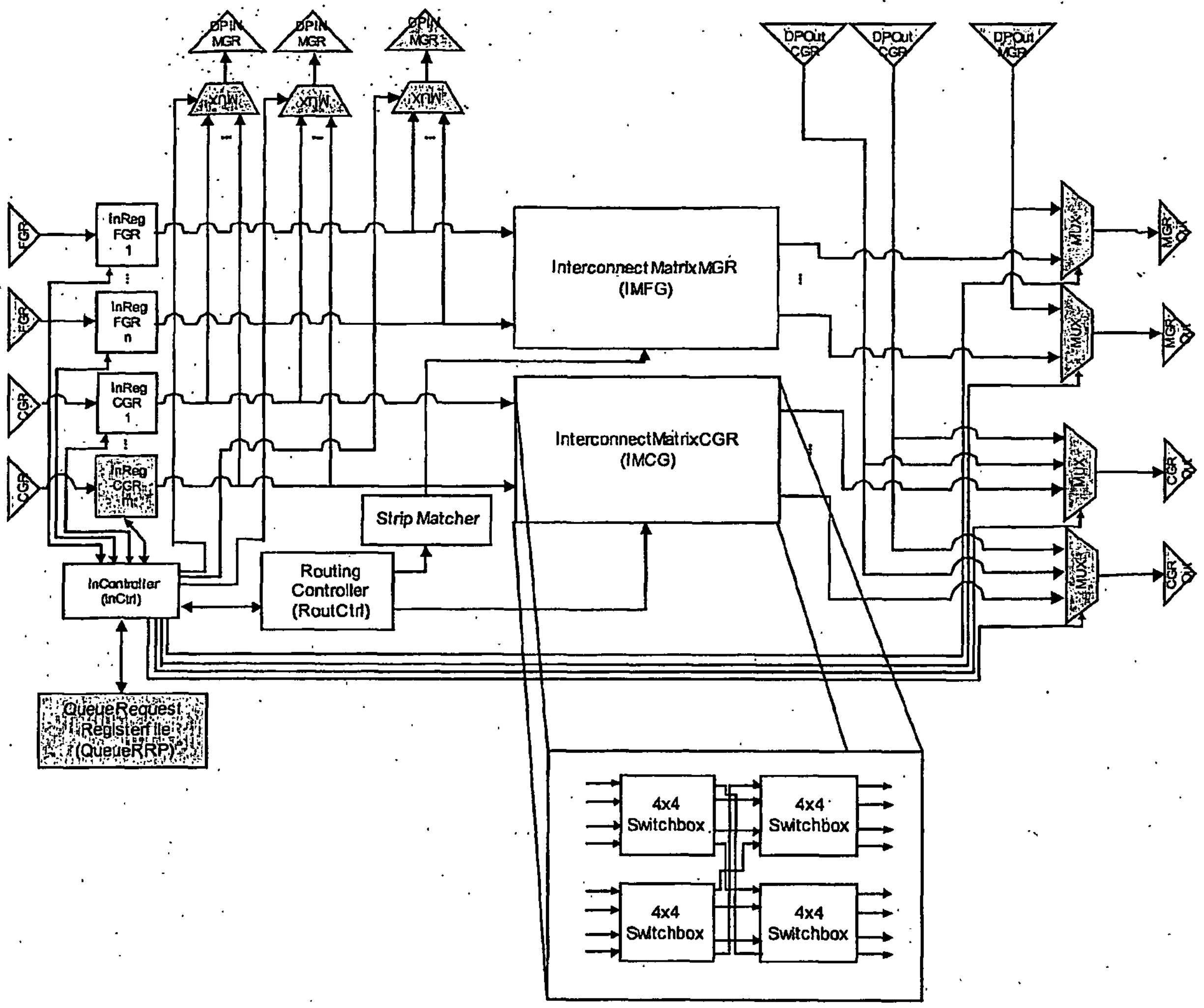

DPIN MGR
DPIN MGR
DPIN MGR
MUX
MUX
MUX
DPOut CGR
DPOut CGR
DPOut MGR
FGR
FGR
CGR
CGR
InReg FGR 1
InReg FGR n
InReg CGR 1
InReg CGR m
Interconnect Matrix MGR (IMFG)
Interconnect Matrix CGR (IMCG)
Strip Matcher
Routing Controller (RoutCtrl)
InController (InCtrl)
QueueRequest Registerfile (QueueRRP)
4x4 Switchbox
4x4 Switchbox
4x4 Switchbox
4x4 Switchbox
MUX
MUX
MUX
MUX
MGR Out
MGR Out
CGR Out
CGR Out

Fig. 42

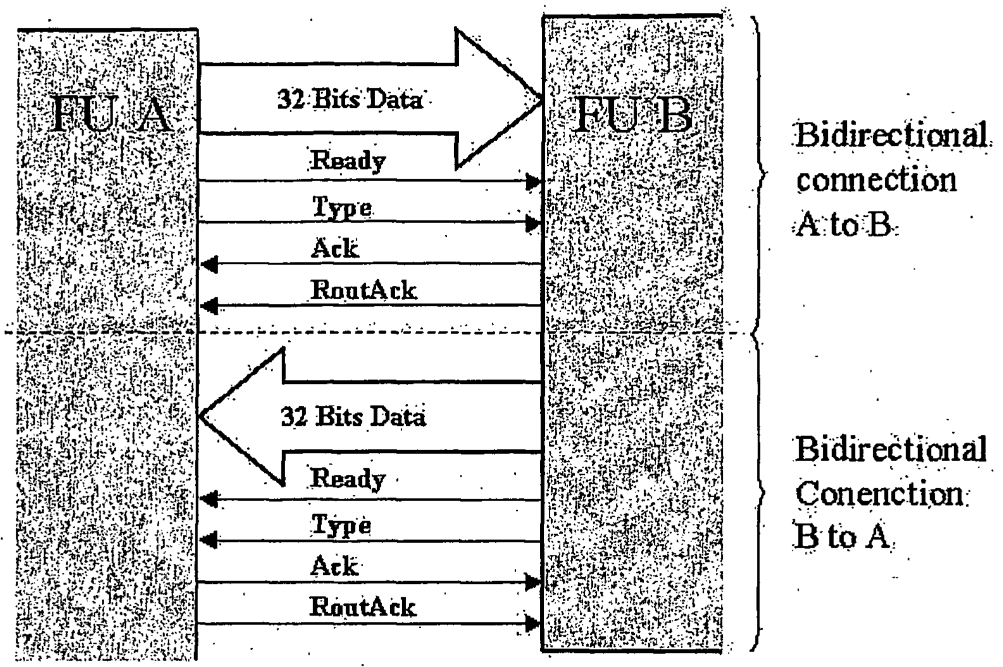
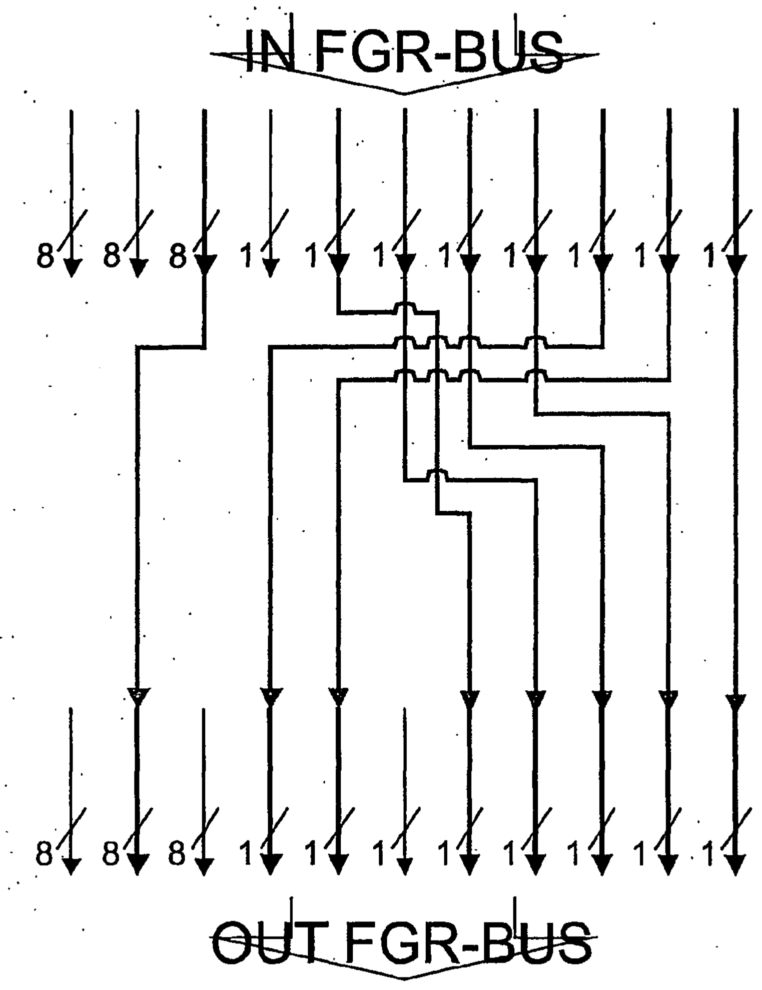

FU A
FU B
32 Bits Data
Ready
Type
Ack
RoutAck
Bidirectional connection A to B
32 Bits Data
Ready
Type
Ack
RoutAck
Bidirectional Conenction B to A

Fig. 43



→ freier Strip mit 1- bzw. 8-Bit Granularität
→ 4-Bit breite logische Verbindung
→ 10-Bit breite logische Verbindung
→ 1-Bit breite logische Verbindung
⇢ freier Strip mit 1- bzw. 8-Bit Granularität

Fig. 44

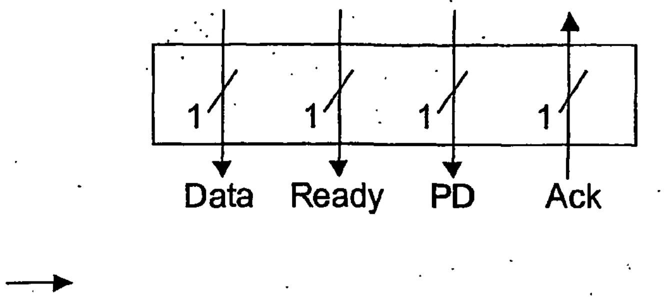


Fig. 45

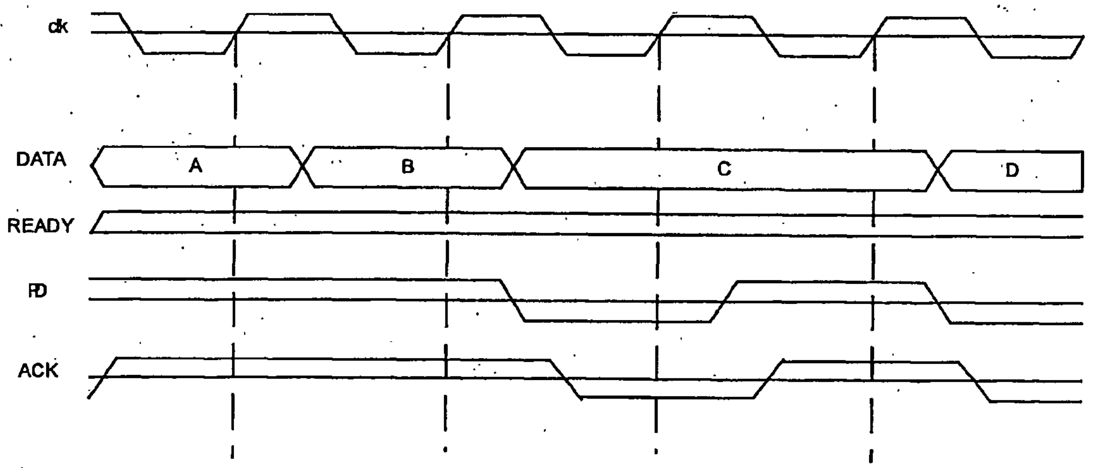


Fig. 46

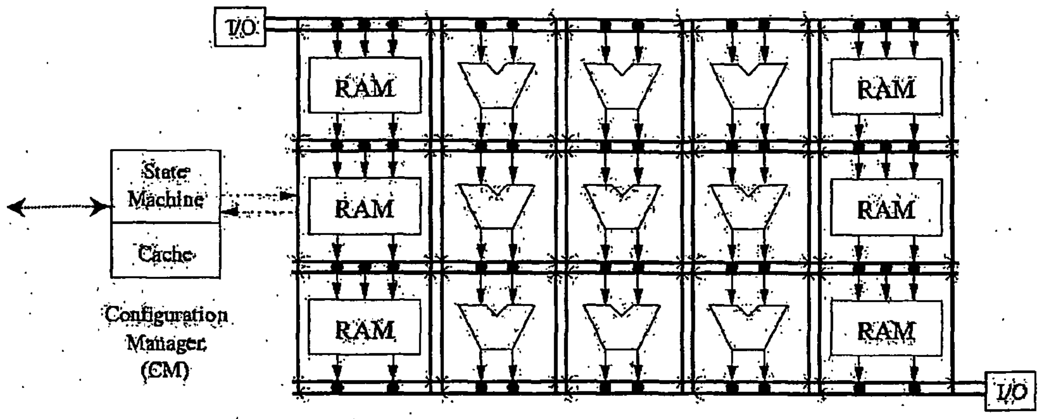


Fig. 47: Simplified structure of an XPP array.

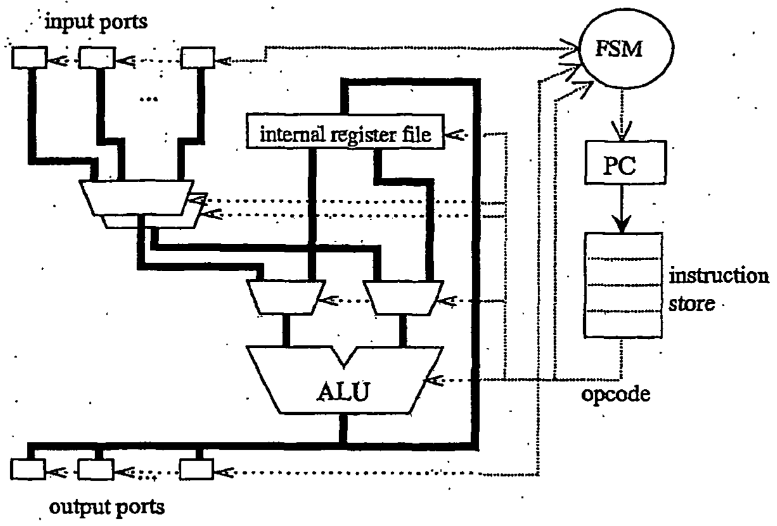
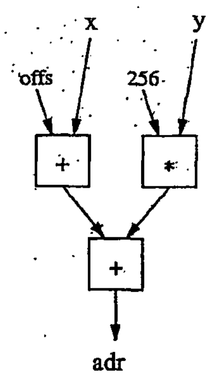
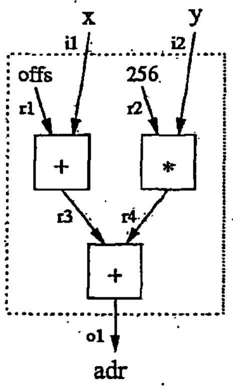


Fig. 48: Function Folding Processing Element



a)



b)

Fig. 49: Address generation data flow graph

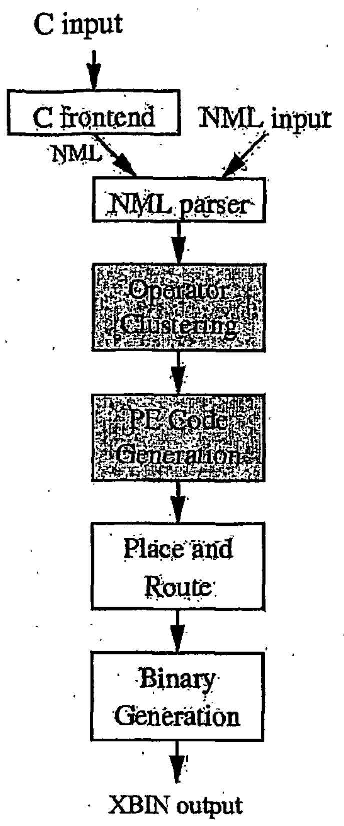
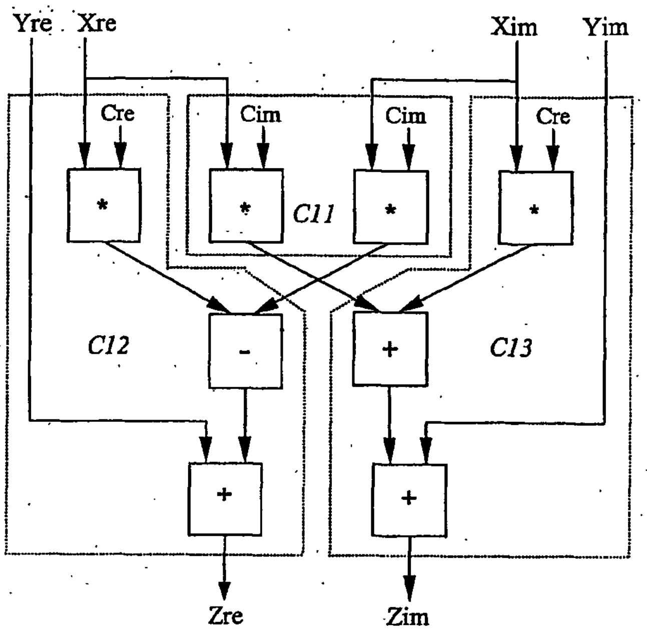
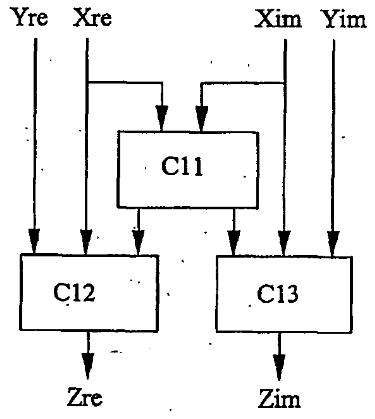


Fig. 50: Extended XPP tool flow.



a)



b)

Fig. 51a und b: Complex FIR filter cell

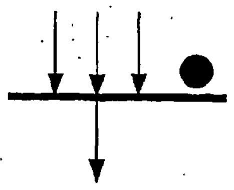


Fig. 52

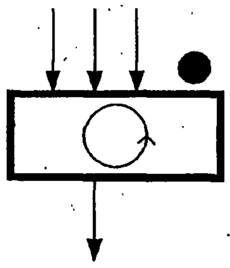


Fig. 53

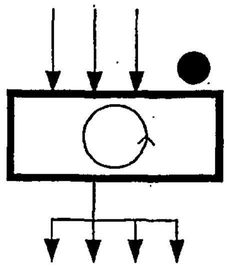


Fig. 54

INTERNATIONAL SEARCH REPORT

International Application No
PCT/EP2004/009640

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F15/80 G06F15/78

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F H03K

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EPO-Internal, WPI Data, INSPEC, COMPENDEX

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category°	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No
X	YEUNG A K W ET AL: "A reconfigurable data-driven multiprocessor architecture for rapid prototyping of high throughput DSP algorithms" SVSTEM SCIENCES, 1993, PROCEEDING OF THE TWENTY-SIXTH HAWAII INTERNATIONAL CONFERENCE ON WAILEA, HI, USA 5-8 JAN. 1993, LOS ALAMITOS, CA, USA, IEEE, US, vol. i, 5 January 1993 (1993-01-05), pages 169-178, XP010640447 ISBN: 0-8186-3230-5	1-4
Y	page 170, right-hand column, paragraph 1 page 173, section 5.3;figure 4 page 174, right-hand column, last paragraph; figure 6	5-7

	-/--	

[X] Further documents are listed in the continuation of box C

[X] Patent family members are listed in annex

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance, the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance, the claimed invention cannot be considered to involve an inventive step when the document is combined with one or *more other* such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search 31 March 2005	Date of mailing of the international search report 08 [illegible] 05
Name and mailing address of the ISA European Patent Office P B 5818 Patentlaan 2 NL - 2230 HV Rijswijk Tel (+31-70) 340-2040 Tx 31 651 epo nl, Fax (+31-70) 340 3016	Authorized officer Kamps, S

II^RNATIONAL SEARCH REPORT

International Application No
PCT/EP2004/0 Θ964 Θ

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No
X	WO 02/071196 A (PACT INFORMATIONSTECHNOLOGIE GMBH; VORBACH, MARTIN; BAUMGARTE, VOLKER) 12 September 2002 (2002-09-12) page 18, lines 2-6; figure 4	1-4
Y	US 2302/138716 Al (MASTER PAUL L ET AL) 26 September 2002 (2002-09-26) paragraph [0058] ; figure 6	5-7
X	KOREN I ET AL: "A DATA-DRIVEN VLSI ARRAY FOR ARBITRARY ALGORITHMS" COMPUTER, IEEE COMPUTER SOCIETY, LONG BEACH., CA, US, US, vol. 21, no. 10, 1 October 1988 (1988-10-01), pages 30-34, XP000118929 ISSN: 0018-9162 page 1; figures 3,4	1-4
A	EP 0 398 552 A (TANDEM COMPUTERS INCORPORATED) 22 November 199Θ (1990-11-22) figure 4B	5-7
A	US 4 918 44Θ A (FURTEK ET AL) 17 April 1990 (1990-04-17) column 6, last paragraph - column 7, paragraph 1; figure 1	5-7
A	BAUMGARTE V ET AL: "PACT XPP - A Self-Reconfigurable Data Processing Archi tecture" NN, 25 June 2001 (2001-06-25), XP002256066 the whole document	1-7
A	HARTENSTEIN R: "Coarse grai n reconfigurable architectures" DESIGN AUTOMATION CONFERENCE, 2001. PROCEEDINGS OF THE ASP-DAC 2001. ASIA AND SOUTH PACIFIC JAN. 30 - FEB. 2, 2001, PISCATAWAY, NJ, USA,IEEE, 30 January 2001 (2001-01-30), pages 564-569, XP010537867 ISBN: 0-7803-5633-6 the whol e document	1-7

INTERNATIONAL SEARCH REPORT

International application No.
PCT/EP2004/009640

Box II Observations where certain claims were found unsearchable (Continuation of item 2 of *first sheet*)

This International Search Report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. [ ] Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely.

2. [ ] Claims Nos.:
because they relate to parts of the International Application that do not comply with the prescribed requirements to such an extent that no meaningful International Search can be carried out, specifically:

3. [ ] Claims Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box III Observations where unity of invention is lacking (Continuation of item 3 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:

see additional sheet

1. [ ] As all required additional search fees were timely paid by the applicant, this International Search Report covers all searchable claims.

2. [ ] As all searchable claims could be searched without effort justifying an additional fee, this Authority did not invite payment of any additional fee.

3. [ ] As only some of the required additional search fees were timely paid by the applicant, this International Search Report covers only those claims for which fees were paid, specifically claims Nos.:

4. [X] No required additional search fees were timely paid by the applicant. Consequently, this International Search Report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

1-7

Remark on Protest

[ ] The additional search fees were accompanied by the applicant's protest.

[ ] No protest accompanied the payment of additional search fees.

FURTHER INFORMATION CONTINUED FROM PCT/ISA/ 210

This International Searching Authority found multiple (groups of) inventions in this international application, as follows:

1. claims: 1-7

The first invention solves the problem of performance enhancement in an multidimensional array of coarse grained logic elements which communicate via busses.
The special technical feature of the first invention is operating the logic elements at a higher clock rate than the busses.

---

2. claims: 8-9

The second invention solves the problem of communicating between coarse grained elements which are arranged in rows.
The special technical feature of the second invention is that the coarse grained element is connected with at least one input to an upper row and with at least one input to a lower row.

---

3. claim: IG

The third invention solves the problem of routing a processing array.
The special technical feature of the third invention is automatically connecting separated fragments of configurations and to rip up nonconnectable traces in a stepwise manner.

IN●ERNATIONAL SEARCH REPORT

Information on patent family membeis

IntelBBonal Application No
PCT/EP2G04/OO9640

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
wo G2071196 A	12-09-2002-	AU 2O6G002 A	22-G4-2Θ2
		AU 8973701 A	05-02-2002
		WO 0213000 A2	14-02-2002
		WO G2Θ964 A2	31-Q1-2002
		WO 0229600 A2	11-04-2002
		WO 02071248 A2	12-09-2002
		WO 02071196 A2	12-Θ-2G02
		WO 02Q71249 A2	12-09-2002
		EP 1377919 A2	07-61-2004
		EP 1342158 A2	10-09-2003
		EP 1348257 A2	O1-10-20G3
		EP 1454258 A2	08-09-20G4
		EP 1386220 A2	04-02-2004
		EP 1540507 A2	15-06-2005
		JP 2004506261 T	26-Θ-2004
		JP 2OO4517386 T	10-06-2094
		JP 2004538675 T	24-12-2004
		JP 2004535613 T	25-11-2004
		JP 2004536373 T	02-12-2004
		us 2004025005 Al	05-02-2004
		us 2004128474 Al	01-07-2004
		us 2OO5066213 Al	24-03-2005
		us 2005086462 Al	21-04-2005
		DE 10227650 Al	19-02-2004
		WO 021Θ532 A2	27-12-2002
		EP 1402382 A2	31-03-2004
		JP 2004533691 T	04-11-2004
		US 2004243984 Al	02-12-2004
		US 2004015899 Al	22-01-2Q04
		CA 2458199 Al	27-02-20Θ
		WO 03017Θ5 A2	27-Θ-2003
		EP 1493084 A2	05-01-2005
		JP 2005508029 T	24-03-2005
		US 2005086649 Al	21-04-2005
		US 2003056202 Al	20-03-2003
		WO 03O23616 A2	20-03-2003
		DE 10297719 D2	17-02-2Θ5
		EP 1449083 A2	25-08-2004
		US 2005022062 Al	27-01-2005
		US 2005053056 Al	10-03-2OO5
		US 2003046607 Al	05-03-2003
		WO 03025781 A2	27-03-2003
		WO 03036507 A2	Θ-Q5-2Q03
		EP 1466264 A2	13-10-2004
		EP 1472616 A2	03-11-2004
		JP 2005515525 T	26-Q5-2OO5
		AU 2003208266 Al	30-07-2003
		WO 03G60747 A2	24-Q7-2003
		DE 10392560 D2	12-05-2005
		EP 1483682 A2	08-12-2004
US 2002138716 Al	26-09-2002	EP 1415399 A2	O6-Q5-2OO4
		JP 2Θ55Θ532 T	31-03-2ΘΘ5
		TW 578098 B	01-Q3-2OO4
		WO 02077849 A2	03-10-2QO2
		US 2Q03Θ4774 Al	20-03-2003
		us 2GG4008640 Al	15-O1-2OO4
		us 2003135743 Al	17-07-2003

INTERNATIONAL SEARCH REPORT

Information on patent family msmbeis

IntelVional Application No
PCT/EP2Θ04/GO9640

Patent document ciiθd in search report		Publication date	Patent family member(s)			Publication date
US 2002138716^	Al	[illegible]	US	2003154357	Al	14-08-2003
		[illegible]	US	2G05091472	Al	28-04-20Θ5
EP 0398552	A	22-11-1990	US	5203Θ5	A	13-04-1993
			AU	638454	B2	01-07-1993
			AU	5470899	A	08-11-1990
			CA	2015853	Al	02-11-1990
			DE	69025795	Dl	18-04-1996
			DE	69Θ25795	T2	22-08-1996
			EP	0398552	A2	22-11-1990
			OP	3069138	A	25-03-1991
			US	5287472	A	15-02-1994
US 4918440	A	17-34-199Θ	AU	8325487	A	θ1-06-1988
			CA	1287122	C	30-07-1991
			DE	3786669	Dl	26-08-1993
			DE	3786669	T2	31-03-1994
			EP	0326580	Al	09-08-1989
			OP	1501671	T	08-06-1989
			OP	2701859	B2	21-01-1998
			PH	30200	A	05-02-1997
			WO	8803727	Al	19-05-1988
			US	5089973	A	18-02-1992
			US	5019736	A	28-05-1991
			US	5155389	A	13-10-1992